USENIX Association

Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '01)

January 29–February 2, 2001
San Antonio, Texas, USA

Conference Organizers

Program Co-Chairs

Rajendra Raj, *Morgan Stanley Dean Witter*
Yi-Min Wang, *Microsoft Research*

Program Committee

Mustaque Ahamed, *Georgia Tech.*
Ken Arnold, *Sun Microsystems*
Don Box, *DevelopMentor*
Murthy Devarakonda, *IBM T.J. Watson Research Center*
Rachid Guerraoui, *Swiss Federal Institute of Technology, Switzerland*
Jennifer Hamilton, *Microsoft Corporation*
Eric Jul, *University of Copenhagen, Denmark*
Doug Lea, *State University of New York at Oswego*
Keith Marzullo, *University of California, San Diego*
Ira Pohl, *University of California, Santa Cruz*
Douglas C. Schmidt, *University of California, Irvine*
Christopher Small, *Sun Microsystems*
Robert Stroud, *University of Newcastle upon Tyne, UK*
Bjarne Stroustrup, *AT&T Labs*
Joe Sventek, *Agilent Laboratories, UK*
Steve Vinoski, *IONA Technologies, Inc.*
Werner Vogels, *Cornell University*
Shalini Yajnik, *Bell Laboratories, Lucent Technologies*
Deborra Zukowski, *Zedak Corp.*

Tutorial Program Chair

Douglas C. Schmidt, *University of California, Irvine*

Advanced Topics Workshop Chair

Murthy Devarakonda, *IBM T.J. Watson Research Center*

The USENIX Association Staff

6th USENIX Conference on Object-Oriented Technologies and Systems

January 29–February 2, 2001

San Antonio, Texas, USA

Wednesday, January 31

Distributed Objects
Session Chair: Werner Vogels, Cornell University

Infrastructure
Session Chair: Murthy Devarakonda, IBM T.J. Watson Research Center

Thursday, February 1

Reflection in Distribution
Session Chair: Doug Lea, State University of New York at Oswego

Programming Techniques
Session Chair: Deborra Zukowski, Zedak Corp.

Proceedings of the

6th USENIX Conference on

Object-Oriented Technologies and Systems

(COOTS '01)

TORBA: Trading Contracts for CORBA

Raphaël Marvie, Philippe Merle, Jean-Marc Geib, Sylvain Leblanc

Laboratoire d'Informatique Fondamentale de Lille
UPRESA 8022 CNRS
Bât. M3 - UFR d'I.E.E.A.
59655 Villeneuve d'Ascq - France
{marvie,merle,geib,leblanc}@lifl.fr

Abstract

Trading is a key function in the context of distributed applications: It allows runtime discovering of available resources. In order to standardize this function, the Open Distributed Processing (ODP) and Object Management Group (OMG) have specified a trading service for CORBA objects: The CosTrading. This specification has two main drawbacks: First, this service is complex to use from applications and second, it does not offer type checking of trading requests at compilation time. Both are discussed in this paper. The main goal of our Trader Oriented Request Broker Architecture *(TORBA) is to provide a trading framework and its associated tools, which tend to offer typed trading operations that are simple to use from applications and checked at compilation time. In that, we define the concept of* Trading Contracts, *written with the* TORBA Definition Language *(TDL). Such contracts are then compiled to generate trading proxies offering simple-to-use interfaces. These interfaces completely hide the complexity of the ODP/OMG CosTrading APIs. In the meantime, TDL contracts could be dynamically used through a generic graphical console exploiting a contract repository. The example used in this paper, clearly states the advantages brought by the TDL trading contracts: type checking at compilation time, simple to use, and providing a powerful framework for CORBA object trading.*

1 Introduction

Nowadays, building, deploying, and running distributed applications rely on a set of services/functions offered by standard middleware like the *Common Object Request Broker Architecture* [19] (CORBA) of the Object Management Group (OMG), the *Distributed Component Object Model* [8] (DCOM) of Microsoft, and more recently the *Java Remote Method Invocation* [24] (RMI) of Sun Microsystems. The main functions of such middleware solutions are synchronous communication using operation invocation, asynchronous communication through message or event passing, transaction monitors, security, persistence, and resource trading. This paper proposes an innovative framework named *Trader Oriented Request Broker Architecture* (TORBA) to trade distributed objects over CORBA.

A middleware trading function tends to provide a means to discover resources available in a distributed system, in order to dynamically interconnect at runtime the various components of an application. For example, it allows a client to find back its associated server. Such a search may be based upon various criteria, like the physical location of the resource (e.g. to find the printer service of the third floor), the symbolic name of the resource (e.g. to find the BestPrint printer), or the characterization of the resource using its properties (e.g. find a color printer faster than ten pages per minute). The conceptual contribution of this paper is to define the concept of *trading contract* in order to characterize both the resource properties, and the search operations used by client applications.

The trading function has been studied both in academic projects and industrial products. Some projects have focussed on the interest of using such a function in a large scale context in order to share resources [16]. In 1993, the ANSA consortium has discussed what the trading function should be [7].

More recently, Sun Microsystems has defined a trading function, included in the *Jini* environment [1]. Based on the easiness with Java to serialize objects, this trading function allows applications to retrieve serialized objects (like network stubs or complete services). Other research works have focussed on traders federations [5], performance [6], and scalability [26]. In the context of object trading, the first technical contribution of this paper is an innovative approach that simplifies and types the use of a trading function.

In order to standardize middleware trading function, the *International Standardization Organization* (ISO) in its *Open Distributed Processing* [11] (ODP) activity and the *Object Management Group* (OMG) have defined a specification of the functional interfaces of such a function [17] using the OMG *Interface Definition Language* (OMG IDL). This specification is mainly based on the work previously performed by the ANSA consortium [7] and the DSTC [28]. It defines a set of generic APIs for applications to export and search CORBA object references in a standard and portable way, whatever the underlying implementation. Unfortunately, these APIs are quite complex to use and very technical. Moreover, using these APIs does not provide trading request type checking at compilation time, but only at runtime. Thus, the second technical contribution of this paper is to perform trading request type checking at compilation time, improving software quality and reliability.

The objective of our work is to define and to offer a typed trading environment being easy to use from CORBA applications. In that, we have defined the *trading contract* concept used to describe typed properties (object characterizations) as well as query operations to be used by applications. The *TORBA Definition Language* (TDL) is used to define these contracts. Then, it is compiled to generate trading proxies offering simple specialized interfaces to be used from client applications. The use of these interfaces is checked at compilation time, based on their types (i.e. operation synopsis). Furthermore, these proxy implementations completely hide the technical complexity of the ODP/OMG trader interfaces. In the meantime, TDL contracts could be stored in a TDL repository, like OMG IDL definitions are stored in CORBA's Interface Repository. Then, this repository could be used dynamically from a graphical console to discover available trading offers and to use defined query operations.

Section 2 of this paper presents an overview of the ODP/OMG CosTrading service. This overview focuses on trading offer typing and use of the query operation, in order to outline their drawbacks: technical complexity and lack of type checking. Section 3 discusses the *trading contract* concept, the TDL language, the proxy generation and execution process, as well as the dynamic approach. It also presents the implementation of TORBA, using a printer service as example to underline the benefits of our approach. Since TORBA use has only been performed using simple examples, section 4 presents some empirical results. Section 5 discusses the related work in middleware that are used in TORBA: the proxy concept, the structure of ORBs, and the component-oriented approach. Finally, section 6 summarizes this paper, in progress, as well as fore-coming work directions.

2 The CosTrading Service

2.1 Overview

The ODP/OMG CosTrading service is similar to a search engine for CORBA object service references. Figure 1 presents the CosTrading standard use, composed of four steps. (1) Service designers define their service offer types (see section 2.2). (2) Service providers or application servers characterize and export their service offers using properties describing the service. (3) Service users or client applications search service references using criteria describing their requirements. (4) Once references have been retrieved, clients invoke operations on the services. All these requests—definition, export, lookup, and use—are carried by CORBA.

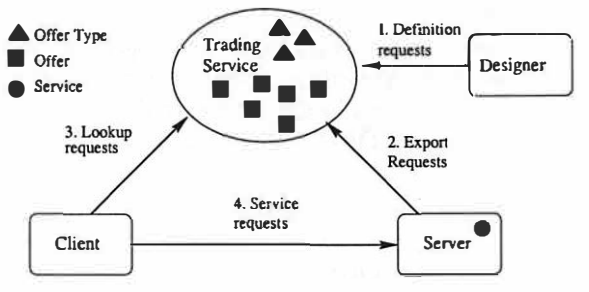


Figure 1: The ODP/OMG CosTrading Service Use.

The CosTrading service provides three main in-

terfaces for applications. The `ServiceTypeRepository` interface is used to define and manage service offer types. The `Register` interface is used to export service offers. Finally, the `Lookup` interface is used to search exported service offers. Other interfaces are also available for administration purposes, like to set the behavior of the trader and its search operation, as well as to build trader federations—offering a potentially large-scaled and unified trading service [2].

2.2 Service Offer Types

In the CosTrading, service offers are strongly typed. Any export operation is based on the use of an offer type, similarly lookup operations are performed upon a given type. Typed offers bring two main advantages. First, applications cannot export or search weird offers, but only offers defined at design time. Second, a CosTrading implementation may take advantage of types to improve performance, only searching in offers of a given type instead of in all the offers. This becomes vital when several thousand of offers have been exported. Part of the CosTrading service, the **Service Type Repository** stores the various service offer types. It also provides type checking when exporting or searching offers. In this repository, a service offer type is characterized using four elements:

- a **name**, which is a unique global identifier in a trader federation and used to define, export, and search service offers,
- some inherited **super types**, used with particular rules for redefinition which are not discussed here,
- an **OMG IDL interface** to which exported service references have to conform, and
- some **service properties** characterizing the exported service.

Each service property is characterized using:

- a **name**, which is also a unique identifier in a service type,
- an OMG IDL **type** which characterizes the type of the property values, and
- a using **mode**, which has to be set to:
  - *normal*: Giving a value to such a property is optional at creation time. If a value is given, it could be modified or removed during execution by the service provider.
  - *readonly*: It is not required to give a value to such a property, but if so the value cannot be modified.
  - *mandatory*: The service provider has to give a value to such a property at exportation time.
  - *mandatory readonly*: The provider has to give a value to the property, which cannot change during the execution.

Figure 2 presents the OMG IDL interface of a printer service. This service is used in this paper to illustrate various aspects of the trading service and our TORBA proposal.

```
interface PrinterServer {
    void print (in string filename) ;
};
```

Figure 2: OMG IDL of a Simple Printer Service.

The related service offer type is Printer, which is characterized by the four properties presented in Table 1. The `color` property specifies if a printer could print in color or only in B&W. The `cost_per_page` property contains the cost to print a sheet of paper for this printer. The number of pages per minute a printer can produce is contained in the `ppm` property. Finally, the `queue` property is the name of the printer queue.

name	type	mode
color	boolean	normal
cost_per_page	float	normal
ppm	unsigned short	normal
queue	string	normal

Table 1: Printer Service Offer Properties.

As stated earlier, it is important to have typed offers. However, dealing with software quality, the CosTrading service lacks a standard language to describe offer types. The only available means is to use the `add_type()` operation of the `ServiceTypeRepository` interface provided by the CosTrading service. Section 3.2 discusses how the

```
module CosTrading {
  interface Lookup : TraderComponents,
                     SupportAttributes,
                     ImportAttributes {
    void query (
      in ServiceTypeName  type,
      in Constraint       constr,
      in Preference       pref,
      in PolicySeq        policies,
      in SpecifiedProps   desired_props,
      in unsigned long    how_many,
      out OfferSeq        offers,
      out OfferIterator   offer_itr,
      out PolicyNameSeq   limits_applied
    ) raises (
      IllegalServiceType,
      UnknownServiceType,
      IllegalConstraint,
      IllegalPreference,
      IllegalPolicyName,
      PolicyTypeMismatch,
      InvalidPolicyValue,
      IllegalPropertyName,
      DuplicatePropertyName,
      DuplicatePolicyName
    );
  };
};
```

Figure 3: The Lookup Interface to Search Offers.

TORBA Definition Language (TDL) addresses this problem.

2.3 Searching Service Offers

As this paper focuses on the search process, the drawbacks of the export process are not discussed here. However, these drawbacks are similar to those presented in this section.

Once offers have been exported by servers, their references and properties could be retrieved using the CosTrading search operation. Figure 3 presents its Lookup interface used to perform searches. The query operation allows clients to find back services from the set of exported offers. The argument number of this operation is quite high. This is due to the genericity required by the operation in order to be usable in a wide number of applications.

The type parameter defines the offer type required by the client application. The constr parameter contains a constraint to be matched by the properties of selected offers. This constraint is a string containing a boolean expression written using the *OMG Constraint Language* (OCL). The pref parameter specifies the returned offer order in OCL. The policies parameter specifies the strategies to be used during the search. The desired_props argument contains the properties to be returned for each offer to the client: none, all, or only specific ones. As there may be a huge number of matching offers, the how_many argument fixes the maximum number of offers to be returned. Following offers could be retrieved later on using the offer_itr iterator provided by the operation. Finally, the two last out parameters contain, after processing, the result (offers) and the limits effectively applied to the search policy (limits_applied).

Furthermore, when providing wrong parameter values, the query operation raises one of the ten listed exceptions. Such exceptions mean a misuse of the CosTrading service related to search strategies or to its type model, like an illegal or unknown type name, a badly expressed preference or constraint, or an unknown property name. Thus, the CosTrading service only checks requests at runtime, while type checking could be performed at application compilation time, improving both software quality and service performance—avoiding runtime type checking. At the moment, TORBA, as discussed in the following, mainly addresses type checking at application compilation time.

Figure 4 presents how an application, written in OMG IDLscript[1] [21], may retrieve offers about color printers faster than two pages per minute. The offers, iter et limits variables are initialized to receive the query() operation results. The offer type, property constraints, and the result order are provided as strings. Thus, it is up to the CosTrading server to check and to evaluate these strings in order to perform the search, implying type errors to be only discovered at runtime.

The simplicity of this excerpt relies on the use of the OMG IDLscript language. However, a real application has to set the search strategy, catch and process the potential exceptions, and process the returned results. The latter includes the offers sequence processing, and potentially the use of the iter iterator to process the following offers. Thus, about fifty lines of Java or C++ are required only to ob-

[1] IDLscript is the CORBA 3.0 scripting language, contribution and specification of our project CorbaScript [4, 13, 14].

```
# variables to receive answers
# using the out mode
offers = Holder()  # returned offers
iter   = Holder()  # next offers iterator
limits = Holder()  # limits applied
trader.query (
  "Printer",       # offer type required
  # OCL for offer constraint
  "color == TRUE and ppm > 2",
  "first",         # answer order
  # use default strategy
  CosTrading.PolicySeq(),
  # properties to be returned
  CosTrading.Lookup.SpecifiedProps (
    CosTrading.Lookup.HowManyProps.some,
    ["queue", "color",
     "cost_per_page", "ppm"]
  ),
  # max number and out params
  100, offers, iter, limits
)
```

Figure 4: Searching offers using IDLscript.

tain the list of color printers faster than two pages per minute. In that, we claim that the query operation is complex and very technical to use. Moreover, the huge use of this operation forces applications to build, invoke, and process many trading requests, introducing code complexity and potential runtime errors. Section 3.3 discusses how TORBA automates this trading related technical code production to simplify application code.

2.4 Review

More than presenting the main operations provided by the ODP/OMG CosTrading service to type and search offers, this section outline the drawbacks of these functions. First, the CosTrading service does not provide a definition language to define offer types. Such a language is mentioned in the CosTrading specification, however only for an illustrative purpose. The service only relies on the use of a type repository used at runtime. Then, the technicity and complexity of this service have been discussed. In order to benefit from the CosTrading service, it is necessary to master the use of operations like query and data structures provided by the service. Finally, using strings to manipulate properties implies runtime type checking and forbids type checking at compilation time. This reduces the easiness to produce reliable applications in an efficient way. To summarize, as any CORBA service, the CosTrading service only offers a set of complete OMG IDL interfaces. This brings the following four questions.

- How to simplify the use of the CosTrading service?
- How to provide type checking at compilation time?
- Which language should be used in order to define offer types?
- Which framework should be applied to trading?

Looking at today's software industry, three answers arise. First, a GUI could be provided to use the trading service easily. This solution is already available for many trading service products. Nevertheless, this choice does not address the use of the trading service from an application. Then, a library may hide the trading service complexity. However, providing such a library is a huge task: It would be easy to suffer the same drawbacks as the CosTrading interfaces. Moreover, it would only define a programming framework, but no design method, nor language to specify offer types. Finally, a trading function, to be specialized to each application needs, could be defined using the concept of *trading contract* as discussed in the following section.

3 The TORBA Proposal

3.1 The Trading Contract Approach

The objective of TORBA is to provide a simple and strongly typed trading facility for CORBA applications. In that, TORBA is based upon the ODP/OMG CosTrading service, taking full advantage of its functionalities like available implementations, complex lookup algorithms, offer persistence, large-scaled trader federations.

Then, the conceptual benefit of TORBA is to define the concept of *trading contracts*. Such a contract is defined at application design time like OMG IDL interface contracts are defined [9]. These contracts take into account offer provider needs as well as client application ones: This results in the definition

of trading offer types. First, offer types clearly identify and group together properties (i.e. name and type of the values) characterizing exported CORBA objects conform to a given OMG IDL interface. Second, offer types also contain a list of query operations commonly used in client applications. Such operation is characterized through a synopsis (name and parameters), as well as a boolean constraint to be applied on both parameters and the properties of the associated type. Offer types may be classified using multiple inheritance. Such a classification permits designers to define abstract types, like a device, that could be specialized to concrete types, like a scanner and a printer. Moreover, concrete types can also be inherited to define new query operations exactly meeting requirements of client applications. Using multiple inheritance improves the reuseness of properties and query operations.

The technical benefit of TORBA is to provide a complete generation and execution environment to use trading contracts. Offer types are defined using the *TORBA Definition Language* (TDL). Such definitions are then compiled to generate trading proxies offering to applications easy-to-use OMG IDL interfaces. The use of these specialized interfaces is thus checked at application compilation time. Moreover, proxy implementations fully hide the ODP/OMG CosTrading technicity. Such implementation is generated for several programming languages: OMG IDLscript, Java, and C++ later on. In the meantime, trading contracts could also be used dynamically through a generic graphical console. Trading contracts are stored into a repository and browsed by the console which can also invoke query operations defined for the given type.

3.2 The TORBA Definition Language

The *TORBA Definition Language* (TDL) is the formalism to define TORBA trading contracts. Using simple typed constructions, it describes offer types, their inheritance relation, their properties (name and type), as well as query operations (name, parameters, and constraints). Property and parameter types rely upon the OMG IDL type model. Constraints are defined using the OMG Constraint Language (OCL) extended to take into account query operation parameters as well as to offer composition of query operations. TDL is defined as two languages: an XML DTD and a BNF grammar. This paper only describes the second one, being more concise and quite familiar to CORBA users. Figure 5 presents an example of offer type definitions.

```
abstract offer Device {
  property string name ;
  query all () is TRUE ;
};
offer Printer : Device {
  interface PrintService ;
  property boolean        color ;
  property float          cost_per_page ;
  property unsigned short ppm ;

  query colors () is color == TRUE ;
  query faster (in unsigned short s)
    is ppm > s ;
  query faster_colors (in unsigned short s)
    is colors () and faster (s) ;
};
```

Figure 5: Trading Offer Type Definition using TDL.

A trading offer type is defined using the `offer` keyword followed by the type name, and possibly the list of inherited super-types. Basically, a type is concrete: provider could export offers using this type. Then, it has to include an `interface` entry defining the base interface to be supported by exported objects. The `abstract` keyword defines a type as being abstract, no offer may be exported for this type. It will be inherited to define concrete types. The TDL contract of Figure 5 defines two offer types related to the printer example of this paper: The `Printer` concrete type inherits from the `Device` abstract type, and specifies offers for objects implementing the `PrintService` interface (or one of its sub-interfaces). Properties are defined using the `property` keyword followed by an optional access mode, an OMG IDL type, and a formal name. If undefined the access mode is `normal` (see section 2.2).

The `Printer` offer includes the four following properties: the `name` string inherited from `Device`, the `color` boolean, the `cost_per_page` float, and the `ppm` unsigned short. Search operations are defined using the `query` keyword followed by a name, potentially a list of arguments (defined as for OMG IDL operations), and a constraint. The constraint is based upon the properties of the offer type (e.g. the `colors()` query), the properties and the parameters (e.g. the `faster()` query), or a composition of query operations (e.g. the `faster_colors()` query). The `all()` query is defined with `TRUE` as constraint

in order to retrieve all the available offers for the `Printer` type. Query operation inheritance has the following semantic: The constraint is kept, however it does not apply on the super-type, but on the inherited type. The operation implementation is implicitly overloaded in generated proxies. When applied to the `Device` type, the `all()` operation returns all the available device offers. When applied to the `Printer` type, it only returns the available printer offers.

Defining specific queries for given values of properties should not be misused. The point is not to define a query for any potential property value, but to define the most commonly used queries. For queries that may appear from time to time only, the generic query operation available with all types has to be used (see section 3.3.1). Nevertheless, using the generic `query()` generated by TORBA already brings type checking and reduces the technicity of the lookup mechanism.

Two constraints are implied by the use of TDL contracts. First, like OMG IDL contracts, TDL contracts have to be globally known to clients. Moreover, the type hierarchy of TDL may be extended but has to stay consistent, i.e. no TDL contract should be removed nor modified. Second, each TDL contract has to be defined using an identifier being unique in the whole system. Here too, like OMG IDL definition it is important for designers to define their TDL contracts using modules in order to avoid name collisions.

This section has presented the second TDL formalism (BNF grammar), being simple to learn. This basis will be extended according to the need arising from our experiments. As an example, dynamic properties specification, whose values are computed at runtime and not statically set at exportation time, seems an interesting extension. However, it is important for this language not to become too complex and underused due to this complexity.

3.3 Trading Proxy Generation

Once trading contracts have been defined using TDL, they may be compiled to generate trading proxies for applications, as depicted in Figure 6.

The TDL compiler checks both the syntax and the semantic of TDL definitions. Semantic checking controls OMG IDL type correctness, TDL type names and properties, as well as OCL constraints in order to ensure no type related problem could arise at runtime. Proxy OMG IDL interfaces are produced by the TDL compiler, as well as their implementation for a given language—IDLscript and Java for the moment, C++ later on. For portability purpose, the TDL compiler is written in the Java language, based on the lexical and syntactic analyzer generated using JavaCC [15].

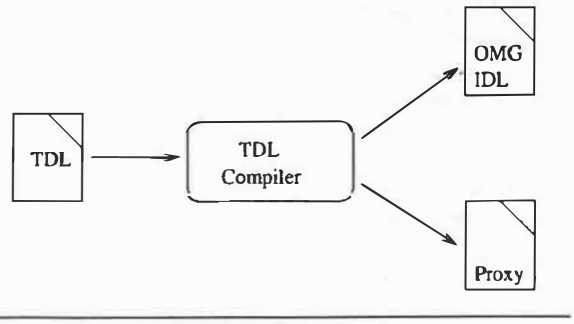


Figure 6: TDL Language Compilation Process.

3.3.1 Generated OMG IDL Interfaces

Each definition of trading offer type is mapped to an OMG IDL module named as the offer type and containing the five following definitions.

- The `Offer` structure represents a trading offer. It contains the exported object reference and a field for each property defined in the offer type or its super-types. Field types are those of the service interface and property types as defined in the TDL offer.
- The `OfferSeq` sequence is used by query operations to return matching offers.
- `OfferType`, `Export`, and `Lookup` interfaces respectively describe the **Service Type Repository** access, the export and the lookup proxies. The latter inherits from the `TORBA::Lookup` interface and contains an operation for each query definition. Its also contains a generic but nonetheless typed query operation.

Figure 7 presents an excerpt (the lookup proxy interface) of the OMG IDL definitions generated for the `Printer` trading contract as defined in Figure 5.

```
#include <TORBA.idl>
module Printer {
  struct Offer {
    PrintService   service ;
    string         name ;
    boolean        color ;
    float          cost_per_page ;
    unsigned short ppm ;
  };
  typedef sequence<Offer> OfferSeq ;

  interface Lookup : TORBA::Lookup {
    OfferSeq query_all () ;
    OfferSeq query_colors () ;
    OfferSeq query_faster
      (in unsigned short s) ;
    OfferSeq query_faster_colors
      (in unsigned short s) ;
    OfferSeq query (in TORBA::Query q)
      raises (TORBA::IllegalConstraint) ;
  };
  // interfaces for type definition
  // and exportation
};
```

Figure 7: OMG IDL Module Generated from the Printer TDL Contract (excerpt)

The `Printer` offer type is mapped to the `Printer` OMG IDL module. The `Offer` structure represents a printer offer. It contains a field for the exported print service, as well as for the `name`, `color`, `cost_per_page`, and `ppm` properties. The lookup proxy `query_all()`, `query_colors()`, `query_faster()`, and `query_faster_colors()` operations represent the queries defined in the `Printer` contract. Parameters are the same as those defined in the contract, while their return type is a printer offer sequence (i.e. `OfferSeq`). The last `query()` operation allows applications to perform searches not defined in the TDL contract. The `TORBA::IllegalConstraint` exception may be raised at runtime if the constraint is malformed.

Experiments have been performed using generation rules presented here, validating these choices. As an example, the `Offer` structure is a good means to perform checking of export and lookup operations at compilation time. However, we also intend to experiment the use of *valuetypes*[2] instead of the structure, as well as using a typed iterator interface instead of the sequence.

[2]Since CORBA 2.3, *valuetypes* permit argument objects to be passed by value instead of by reference.

```
# File 'PrinterProxies.cs'
import TORBArt
class Lookup (TORBArt.LookupBase) {
  proc __Lookup__ (self) {
    self.__LookupBase__ ("Printer",
                         Printer.Lookup)
  }
  proc query (self, constraint) {
    answers = []
    for offer in
      self.generic_query (constraint) {
      answers.append (
        Printer.Offer (
          offer["service"],
          offer["name"], offer["color"],
          offer["cost_per_page"],
          offer["ppm"]
      ) )
    }
    return answers
  }
  proc query_faster (self, s) {
    return self.query ("ppm >= " +
                       s._toString())
  }
  proc query_all (self) {
    return self.query ("TRUE")
  }
  # other query operations
}
```

Figure 8: OMG IDLscript implementation of the Printer lookup proxy (excerpt)

3.3.2 Generated Proxy Implementation

The generation of proxy implementations depends on the constructions of a given language. However, using an object-oriented language, each OMG IDL interface is implemented by a class inheriting from a base class provided by the TORBA runtime. These classes fully hide the ODP/OMG CosTrading technicity: use of the service interfaces and data structures, as well as exception handling. Such runtime classes provide generic operations used from proxy implementations. Figure 8 presents an excerpt of the lookup proxy implementation for the `Printer` offer type, generated for the OMG IDLscript language.

The `Lookup` class inherits from the `TORBArt.LookupBase` class provided by the TORBA runtime. The `__Lookup__` constructor invokes the super-class constructor providing the TDL type name (i.e. `Printer`), as well as the im-

```
lookup = PrinterProxies.Lookup()
offers1 = lookup.query_faster_colors (2)
offers2 = lookup.query ("color == FALSE
  and cost_per_page < 0.05 and ppm > 10")
```

Figure 9: Printer Search Proxy Use.

plemented interface type (i.e. `Printer::Lookup`). The generic query operation is invoked by the `query` operation providing the constraint to apply. Then, the result is translated to a printer offer sequence (i.e. `Printer::OfferSeq`). The implementation of query operations, like `query_faster` and `query_all`, only consists of creating the associated constraint and invoking the `query` operation.

3.4 Using TORBA Proxies

Figure 9 presents, in OMG IDLscript, the use of lookup proxy presented in the previous section. The first line instantiates the lookup proxy class. The second line invokes the query operation to find color printers faster than two pages per minute. This operation realizes the same search processing as the one presented in Figure 4. Simplicity brought up by TORBA becomes clear. The application developer does not bother with the trader technicity, he/she can focus on the use of the trading contract only. Moreover, the operation execution cannot fail as types have been checked by the TDL compiler. It can only return an empty sequence if no offer matches the search. The third line illustrates the option of using a search operation not defined in the trading contract: Searching offers related to B&W printers faster than ten pages per minute, for a cost less than five cents a page. Nevertheless, even if the use of this operation is provided, software engineering quality is improved when all the search requests are defined in the trading contract.

3.5 Execution of TORBA Proxies

Figure 10 presents the set of objects involved during the execution of a query operation. The lookup proxy object and its CORBA stub are co-located with the application. This latter invokes the proxy operations through its OMG IDL interface. The proxy operation implementation invokes the TORBA runtime class providing the appropriate constraint. Then, this class invokes the CORBA stub providing access to the ODP/OMG CosTrading service. As a result, the runtime class catches the exceptions and the proxy class translates data from its CosTrading representation to the representation defined in the trading contract. One future work is to measure the overhead of lookup proxies and to optimize their implementations in order to be close to native CosTrading performance (see section 4).

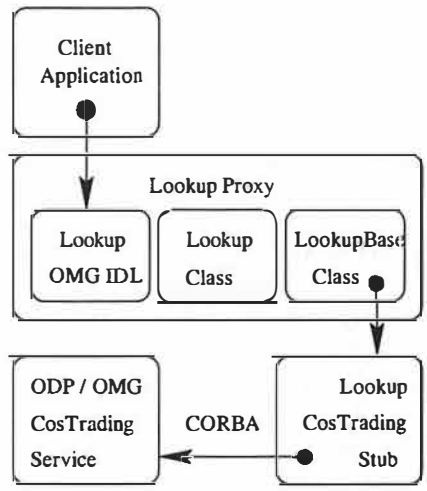


Figure 10: Execution Process of Lookup Proxy Operations.

3.6 The TORBA Dynamic Approach

Previous sections have presented the conceptual benefits of TDL contracts, as well as the technical ones brought by generation and execution of related proxies. In the meantime, the TORBA environment offers a dynamic approach to use trading contracts as depicted in Figure 11. This approach permits one to build applications without static knowledge, at design time, about used trading contracts. This knowledge will be learnt at runtime.

The dynamic approach in TORBA relies on a trading contract repository. This repository, currently written in OMG IDLscript, stores TDL contracts as a graph of CORBA objects. Each object of the graph represents at runtime a semantic construction of the TDL language. Thus, `offer`, `property`, and `query` constructions are mapped to `OfferDef`, `PropertyDef`, and `QueryDef` interfaces defined in the `TORBA` module. These interfaces provide operations to create and browse objects of the graph,

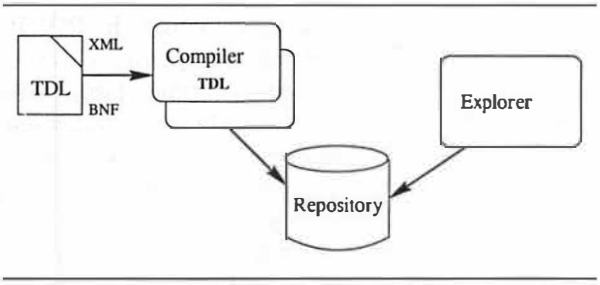


Figure 11: The TORBA Dynamic Approach.

providing TDL information at runtime. Creation operations are used by a specific version of the TDL compiler in order to feed the repository. Other operations are used by any TORBA application requiring dynamic discovering of available trading contracts. In order to validate this approach, we have realized in JavaIDLscript[3] a first dynamic application: the TORBA explorer, illustrated in Figure 12.

Through a GUI written using Java Swing, the TORBA explorer allows users to browse available trading contracts, to select a contract, to consult associated offers, and to perform predefined or specific query operations. The explorer implementation does not rely on any trading contract: Graphical interfaces are dynamically built at runtime according to trading contracts discovered into the TORBA repository. Thus, the TORBA explorer provides a trading GUI dedicated to the contracts used by applications, unlike GUI included with CosTrading implementations. Finally, this explorer is a generic and graphical proof of the relevance and strength of the trading contract concept presented in this paper.

4 Empirical Results

TORBA introduces extra processing while sending requests to the CosTrading. This implies an overhead. However, in the context of distributed applications, there are two levels in the evaluation of overhead. First, there are remote method invocations which overhead is potentially high. Second, there are local method invocations which overhead is most of the time insignificant compared to remote method one. In the context of TORBA, the overhead is introduced by the use of a local library, which means local invocations only : Three local invocations are added for each trading request. Thus, it just increases local processing time and keeps the number of remote method invocations identical, compared to the standard use of the CosTrading. Then, based on early test performed using the ORBacus Trader, the overhead introduced by TORBA is, without any ORB specific optimizations in producing TORBA proxies, less than 5%.

There are few ways to improve performance related to trading using TORBA. First, if the only use of the trader is performed through TORBA, then most of the trader checks (like type checking) can be removed since already performed by the proxies. Thus, the overhead brought up by the proxy would be balanced. Second, proxies could be located close to the trading server and not close to the client. Then, the number of network requests could be optimized, improving global performance. Third, using *smart proxies*, local to the client, to perform caching of trading results, the global performance could be optimized. Finally, a composition of the three aspects will bring the best results. Points two and three are not incompatible as proxies would be split between the trading server and the client application. Client side proxies would perform caching while server side ones would be dedicated to type checking and network optimization.

5 Comparison and Source of Inspiration

During the 80's, the ODP community has defined a type management system for the ODP trading function [10]. This research result has been partly integrated in the ODP / OMG CosTrading specification. However, to our knowledge, no similar works to our TORBA proposal have been performed to reduce the CosTrading complexity and to increase reliability of trading based applications. The trading aspects of ODL (Object Definition Language) defined by the TINA consortium [22] were only related to defining trading properties of objects. This could only be seen as basic trading contracts : attributes are inevitably strings and ODL do not permit to define typed queries. Thus, the complexity of the trader's use is not reduced actually. In the meantime, our original work relies on the use of well-known mechanisms of distributed object computing

[3] JavaIDLscript is our second implementation of the OMG IDLscript language offering access to CORBA and Java objects in the meantime.

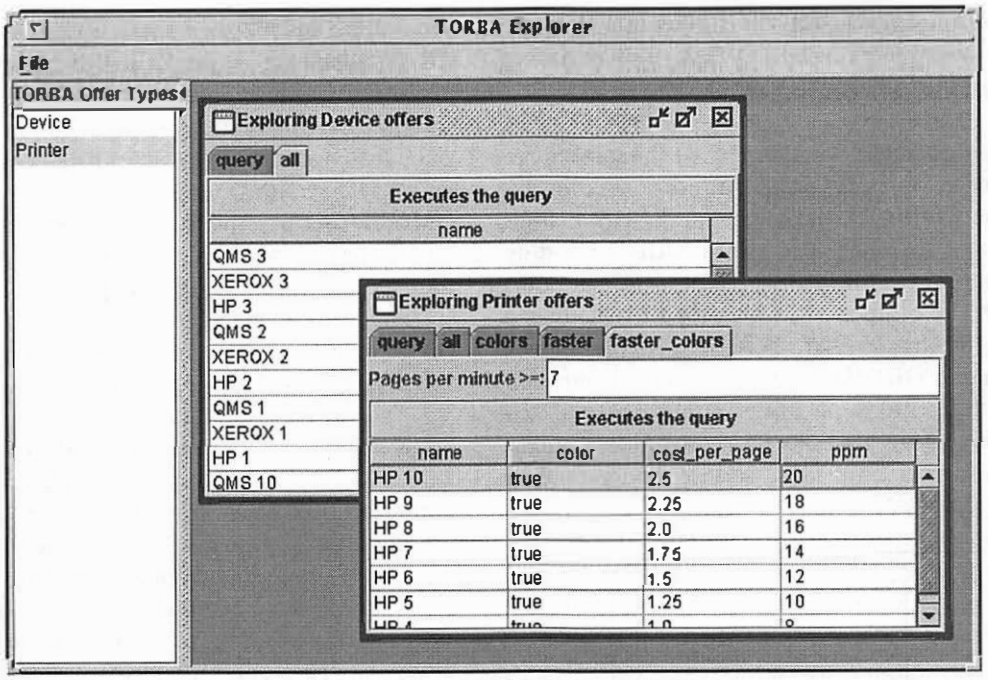


Figure 12: The TORBA Explorer.

middleware: the proxy principle, the ORB structure, and the component approach.

The proxy principle has been defined in [23] as a structural concept to build distributed applications, acting on the behalf of a remote object. This principle extends the RPC (Remote Procedure Call) mechanism as defined in [3] in order to use it in an object-oriented context (i.e. Remote Method Invocation). At the communication level, a proxy (a.k.a. stub) serializes invocations to remote objects like in CORBA [19], DCOM [8], and Java RMI [24] environments. Such a proxy implementation fully hides the technicity related to the serialization process: marshalling of parameters into a network message, care taking of heterogeneity, network layer and error management, and finally unmarshalling the network reply to application data. These proxies are generated based on communication contracts written using an interface definition language (IDL). These IDL descriptions simplify and bring automation to produce the implementation of communication means, increasing the reliability of applications. In the context of TORBA, the communication contract concept, the IDL language, and communication proxies are transposed to trading contracts, the TDL language, and trading proxies. Thus, TDL descriptions simplify and bring automation to produce code related to trading, increasing application reliability.

Compared to *smart proxies* used in the Quality of Objects (QuO) middleware [30], or as implementation of meta-programming mechanism [29], TORBA proxies cannot be labeled as *smart*. In these two examples, *smart proxies* are proxies that potentially perform more processing—like logging, caching, QoS control, or meta-programming—in a transparent way from the client point of view. Extra processing is added to the standard one, without modifying the proxy interface. In the context of TORBA, proxies have an explicit interface which is different from the classical interface of the CosTrading. Moreover, *smart proxies* tend to offer dynamic mechanism for reconfiguration while TORBA proxies are quite static, and could not be changed dynamically at runtime.

TORBA is close to CORBA. The OMG IDL language permits designers to describe interface contracts for CORBA objects, while the TDL language permits them to define trading contracts. The OMG IDL language is compiled to produce communication stubs, or to feed the Interface Repository. Similarly, the TDL language is compiled to generate trading proxies, or to feed the trading contract

repository. CORBA stubs rely on an ORB runtime, encapsulating the GIOP/IIOP protocol, while TORBA proxies rely on the TORBA runtime hiding the CosTrading service, as well as the ORB.

The component-oriented approach is the last source of inspiration of the TORBA proposal. As an example, the CORBA Component Model [18] defines a component as being a software entity providing multiple interfaces (or facets). Each facet is a point of view on the component, which logically defines a set of operations. In that, TORBA provides access to the generic ODP/OMG trading service through facets dedicated to application requirements. Each lookup proxy generated is a dedicated facet being a point of view on the trading service as depicted in Figure 13.

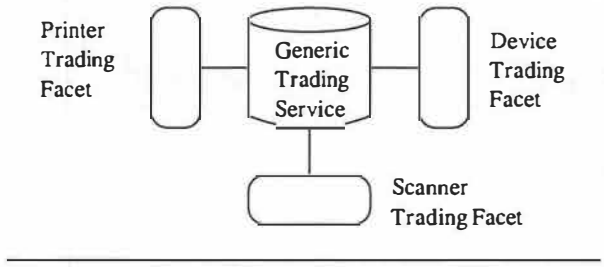


Figure 13: TORBA, towards a 'componentized' trading service.

6 Conclusion

First, this paper has reviewed the ODP/OMG `CosTrading` service. This review has presented the use of the service as being very technical and complex due to the lack of a structured approach. The various drawbacks brought up by the lack of type-checking at compilation time have been underlined. Then, the lack of formalism to define offer types and search operations has been presented as being one of the reasons of the service complexity.

Then, TORBA has been presented as a framework structuring the ODP/OMG trading service use. The conceptual contribution of this paper relies on the definition of the trading contract concept as a paradigm to structure the trading activity. The benefits of the TDL formalism use and its associated tools have been discussed. Using an example, the benefits of TORBA have been illustrated in terms of type checking, simplicity, productivity, and reliability of applications.

All the elements depicted in this paper have been prototyped and experiments have been performed using IDLscript and Java languages, as well as the ORBacus trading service [20]: TDL compilers (BNF and XML versions), proxy generators (OMG IDL, OMG IDLscript, and Java), runtime environments for IDLscript and Java, the trading contract repository, as well as the TORBA explorer are already operational. The next step is to finalize the TORBA environment in order to release it, and to obtain experiment/use feedback from end-users.

From now on, we have lots of work in view around TORBA: (1) support of C++ applications, (2) experiments over other `CosTrading` implementations, (3) measure of the overhead implied by TORBA proxies, (4) experiments of iterators, dynamic properties, and lookup strategies, (5) extension towards asynchronous trading (notification to applications of newly exported offers), and (6) use of the TORBA approach in the context of Jini, trading serialized objects and not only references.

In the meantime, TORBA is part of our actual research work. We intend to use TORBA in order to experiment the concept of **Component Oriented Trading** (COT) [27]. In that, TORBA would become the basis of TOSCA (*Trading Oriented System for Component-based Applications*), whose goal is to provide an environment to deploy and to administrate distributed component based applications [12].

Finally, in a more ambitious vision, we intend to consider the benefits of a language to perform queries and to act upon distributed objects. The goal would be to unify search operations on trading services, object-oriented databases, and object environments *à la* JavaSpaces [25]. This language could be named **TORBA Query Language**, relying upon the following equation:

$$TQL = TDL + OCL + OQL + IDLscript$$

References

[1] K. Arnorld and al. *The Jini Specification.* Addison-Westley, first edition, June 1999. *ISBN: 0-201-61634-3.*

[2] D. Belaid, N. Provenzano, and C. Taconet. Dynamic Management of CORBA Trader Feder-

ation. In *Proceedings of the 4th USENIX Conference on Object Oriented Technologies and Systems (COOTS'98)*, Santa Fee, New Mexico, USA, April 1998. USENIX.

[3] A. Birrell and B. Nelson. Implementing Remote Procedure Call. Technical Report CSL-83-7, Xerox, October 1983.

[4] CorbaWeb. CorbaScript Home Page. URL: http://corbaweb.lifl.fr.

[5] G. Craske and Z. Tari. A Property-based Clustering Approach for the CORBA Trading Service. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, Las Vegas, 1998.

[6] G. Craske, Z. Tari, and K. Kumar. DOK-Trader: A CORBA Persistent Trader with Query Routing Facilities. In *International Symposium on Distributed Objects and Applications (DOA'99)*, Edinburgh, September 1999.

[7] J.-P. Deschrevel. The ANSA Model for Trading and Federation. Technical report, ANSA, July 1993.

[8] R. Grimes. *Professional DCOM Programming.* Wrox Press ltd., Birmingham, Canada, 1997.

[9] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++.* Addison-Westley, 1999. *ISBN: 0-201-37927-9.*

[10] J. Indulska, M. Bearman, and K. Raymond. A Type Management System for an ODP Trader. In *Proc. of the International Conference on Open Distributed Processing (ICODP'93)*, pages 141–152, Berlin, Germany, September 1993.

[11] ISO. *Open Distributed Processing Reference Model – parts 1-4.* International Standard Organization, 1995. ISO 10746-1..4.

[12] R. Marvie, P. Merle, and J.-M. Geib. Towards a Dynamic CORBA Component Platform. In *Proceedings of the 2nd International Symposium on Distributed Object Applications (DOA'2000)*, Antwerp, Belgium, September 2000. IEEE.

[13] P. Merle, C. Gransart, and J.-M Geib. CorbaScript and CorbaWeb: A Generic Object Oriented Dynamic Environment upon CORBA. In *Proceedings of TOOLS Europe 96*, Paris, June 1996.

[14] P. Merle, C. Gransart, and J.-M. Geib. Using and Implementing CORBA Objects with CorbaScript. *Object-Oriented Parallel and Distributed Programming*, 2000. Ed. Hermes.

[15] Metamata. Java Compiler Compiler User Guide. *http://www.metamata.com/javacc/index.html.*

[16] Y. Ni and A. Goscinski. Trader Cooperation to Enable Object Sharing among Users of Homogeneous Distributed Systems. Technical report, RHODOS Project, 1993.

[17] OMG. *CORBAServices: Common Object Services Specification.* Object Management Group, November 1997.

[18] OMG. *CORBA Components: Joint Revised Submission.* Object Management Group, August 1999. OMG TC Document orbos/99-07-{01..03,05} orbos/99-08{05..07,12,13}.

[19] OMG. *CORBA/IIOP 2.3.1 Specification.* Object Management Group, October 1999.

[20] OOC. ORBacus Trader. *http://www.ooc.com.*

[21] OOC and LIFL. *CORBA Scripting - Joint Revised Submission.* Object Management Group, August 1999.

[22] A. Parhar. TINA Object Definition Language Manual v2.3. Technical report, TINA-C, 1996.

[23] M. Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings of the 6th International Conference on Distributed Computing Systems (ICDCS 86)*, pages 198–204, Cambridge, Mass., USA, May 1986. IEEE.

[24] Sun. *Java Remote Method Invocation Specification.* Sun Microsystems, October 1998.

[25] Sun. *JavaSpaces Service Specification.* Sun Microsystems, May 2000.

[26] Z. Tari and G. Craske. A Query Propagation Approach to Improve CORBA Trading Service Scalability. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, Taiwan, April 2000. IEEE.

[27] S. Terzis and P. Nixon. Component Trading: The Basis for a Component-Oriented Development Framework. In *WCOP'99 Proceedings of the Fourth International Workshop on Component-Oriented Programming*, 1999.

[28] A. Vogel, M. Bearman, and A. Beitz. Enabling Interworking of Traders. In *Proceedings of the 3rd International IFIP TC6 Conference on Open Distributed Processing (ICODP'95)*, Brisbane, Australia, February 1995. ChapMan and Hall.

[29] N. Wang, K. Parameswaran, and D. Schmidt. The Design and Performance of Meta-Programming Mechanism for Object Request Broker Middleware. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, San Antonio TX, USA, January 2001. USENIX.

[30] J. Zinky, D. Bakken, and R. Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, 3(1), April 1997.

Dynamic Resource Management and Automatic Configuration of Distributed Component Systems*

Fabio Kon†
Department of Computer Science
University of São Paulo, Brazil
kon@ime.usp.br http://www.ime.usp.br/~kon

Tomonori Yamane
Energy and Industrial Systems Center
Mitsubishi Electric Corporation
yamane@isl.melco.co.jp

Christopher K. Hess Roy H. Campbell M. Dennis Mickunas
Department of Computer Science
University of Illinois at Urbana-Champaign
{ckhess,roy,mickunas}@cs.uiuc.edu
http://choices.cs.uiuc.edu/2k

Abstract

Component technology promotes code-reuse by enabling the construction of complex applications by assembling off-the-shelf components. However, components depend on certain characteristics of the environment in which they execute. They depend on other software components and on hardware resources.

In existing component architectures, the application developer is left with the task of resolving those dependencies, i.e., making sure that each component has access to all the resources it needs and that all the required components are loaded. Nevertheless, according to encapsulation principles, developers should not be aware of the component internals. Thus, it may be difficult to find out what a component really needs. In complex systems, this manual approach to dependency management can lead to disastrous results.

In this paper, we propose an integrated architecture for managing dependencies in distributed component-based systems in an effective and uniform way. The architecture supports automatic configuration and dynamic resource management in distributed heterogeneous environments. We describe a concrete implementation of this architecture and present experimental results.

*This research is supported by the National Science Foundation, grants 98-70736, 99-70139, and EIA99-72884EQ.

†Fabio Kon is supported in part by a grant from CAPES, the Brazilian Research Agency, proc.#1405/95-2.

1 Introduction

As computer systems are being applied to more and more aspects of personal and professional life, the quantity and complexity of software systems is increasing considerably. At the same time, the diversity in hardware architectures remains large and is likely to grow with the deployment of embedded systems, PDAs, and portable computing devices. All these platforms will coexist with personal computers, workstations, computing servers, and supercomputers. The construction of new systems and applications in an easy and reliable way can only be achieved through the composition of modular hardware and software.

Component technology has appeared as a powerful tool to confront this challenge. Recently developed component architectures support the construction of sophisticated systems by assembling together a collection of off-the-shelf software components with the help of visual tools or programmatic interfaces. Components will be the unit of packaging, distribution, and deployment in the next generation of software systems. However, there is still very little support for managing the dependencies among components. Components are created by different programmers, often working in different groups with different methodologies. It is hard to create robust and efficient systems if the dependencies between components are not well understood.

Until recently, highly-dynamic environments with mobile computers, active spaces, and ubiquitous

multimedia were only present in science fiction stories or in the minds of visionary scientists like Mark Weiser [Wei92]. But now, they are becoming a reality and one of the most important challenges they pose is the proper *management of dynamism*. Future computer systems must be able to configure themselves dynamically, adapting to the environment in which they are executing. Furthermore, they must be able to react to changes in the environment by dynamically *re*configuring themselves to keep functioning with good performance, irrespective of modifications in the environment.

Unfortunately, the existing software infrastructure is not prepared to manage these highly-dynamic environments properly.

Existing component-based systems face significant problems with reliability, administration, architectural organization, and configuration. The problem behind all these difficulties is the lack of a unified model for representing dependencies and mechanisms for dealing with these dependencies. Components depend on hardware resources (such as CPU, memory, and special devices) and software resources (such as other components, services, and the operating system). Not resolving these dependencies properly compromises system efficiency and reliability.

As systems become more complex and grow in scale, and as environments become more dynamic, the effects of the lack of proper dependence management become more dramatic. Therefore, we need an integrated approach in which operating systems, middleware, and applications collaborate to manage the components in complex software systems, dealing with their hardware and software dependencies properly.

Software is in constant evolution and new component versions are released frequently. How can one run the most up-to-date components and make sure that they work together in harmony? This requires mechanisms for (1) code distribution over wide-area networks so we can push or pull new components as they become available and (2) safe dynamic reconfiguration so we can plug new components when desired.

In previous papers, we introduced a model for representing dependencies in distributed component systems [KC99] and described a reflective ORB that supports dynamic component loading in distributed environments [KRL+00, KGA+00]. In this paper, we extend our previous work by describing the design, implementation, and performance of an integrated architecture that provides mechanisms for:

1. Automatic configuration of component-based applications.
2. Intelligent, dynamic placement of applications in the distributed system.
3. Dynamic resource management for distributed heterogeneous environments.
4. Component code distribution using push and pull methods.
5. Safe dynamic reconfiguration of distributed component systems.

1.1 Paper Contents

Section 2 gives a general overview of our architecture for automatic configuration and dynamic resource management. Section 3 details the automatic configuration mechanisms, explaining the concepts of prerequisites (Section 3.1), component configurators (Section 3.2), and the Automatic Configuration Service (Section 3.3). Section 4 describes the Resource Management Service, addressing resource monitoring (Section 4.1) resource reservation (Section 4.2), application execution (Section 4.3), and fault-tolerance and scalability (Section 4.4).

Section 5 gives additional implementation details and present experimental results. We then present related work (Section 6), future work (Section 7), and our conclusions (Section 8).

2 Architectural Framework

To deal with the highly-dynamic environments of the next decades, we propose an architectural framework divided in three parts. First, a mechanism for dependence representation lets developers specify component dependencies and write software that deals with these dependencies in customized ways. Second, an Automatic Configuration Service is responsible for dynamically instantiating component-based applications by analyzing and resolving their component dependencies at runtime.

A Resource Management Service is responsible for managing the hardware resources in the distributed system, exporting interfaces for inspecting, locating, and allocating resources in the distributed, heterogeneous system.

Figure 1 presents a schematic view of the major elements of our architecture. *Prerequisite specifications* reify static dependencies of components towards its environment while *component configurators* reify dynamic, runtime dependencies.

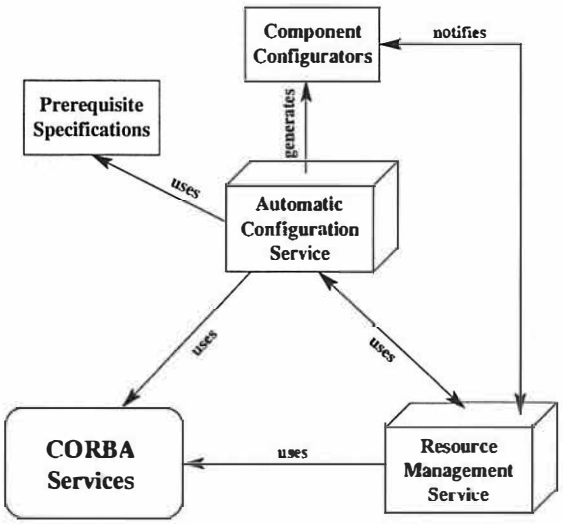


Figure 1: Architectural Framework

As we explain in Section 3, the automatic configuration process is based on the prerequisite specifications and constructs the component configurators. As the Automatic Configuration Service instantiates new components, it uses the Resource Management Service to allocate resources for them. At execution time, changes in resource availability may trigger call-backs from the Resource Management Service to component configurators so that components can adapt to significant changes in the underlying environment.

As described in Section 4.3, when a client requests the execution of an application to the Resource Management Service, the latter finds the best location to execute the application and then uses the Automatic Configuration Service to load the application components.

The elements of the architecture are exported as CORBA services and their implementation relies on standard CORBA services such as Naming and Trading [OMG98].

We have employed the architecture presented here to support a reliable, dynamically configurable Multimedia Distribution System. Readers interested in a detailed description of how our services were used in that particular application scenario should refer to [KCN00]. In the following sections, we provide a more in-depth description of each of the elements of the architecture.

3 Automatic Configuration

Software systems are evolving more rapidly than ever before. Vendors release new versions of web browsers, text editors, and operating systems once every few months. System administrators and users of personal computers spend an excessive amount of time and effort configuring their computer accounts, installing new programs, and, above all, struggling to make all the software work together[1].

In environments like MS-Windows, the installation of some applications is partially automated by "wizard" interfaces that direct the user through the installation process. However, it is common to face situations in which the installation cannot complete or in which it completes but the software package does not run properly because some of its (unspecified) requirements are not met. In other cases, after installing a new version of a system component or a new tool, applications that used to work before the update, stop functioning. It is typical that applications on MS-Windows cannot be cleanly uninstalled. Often, after executing special uninstall procedures, "junk" libraries and files are left in the system. The application does not know if it can remove all the files it has installed because the system does not provide the clear mechanisms to specify which applications are using which libraries.

To solve this problem, we need a completely new paradigm for installing, updating, and removing software from workstations and personal computers. We propose to automate the process of software maintenance with a mechanism we call *Automatic Configuration*. In our design of an automatic configuration service for modern computer environments, we focus on two key objectives:

[1] When even PhD students in Computer Science have trouble keeping their commodity personal computers functioning properly, one can notice that something is very wrong in the way that commercial software is built nowadays.

1. Network-Centrism and

2. a "What You Need Is What You Get" (WYNIWYG) model.

Network-Centrism refers to a model in which all entities, users, software components, and devices exist in the network and are represented as distributed objects. Each entity has a network-wide identity, a network-wide profile, and dependencies on other network entities. When a particular service is configured, the entities that constitute that service are assembled dynamically. Users no longer need to keep several different accounts, one for each device they use. In the network-centric model, a user has a single network-wide account, with a single network-wide profile that can be accessed from anywhere in the distributed system. The middleware is responsible for instantiating user environments dynamically according to the user's profile, role, and the underlying platform [CKB+00].

In contrast to existing operating systems, middleware, and applications where a large number of non-utilized modules are carried along with the standard installation, we advocate a *What You Need Is What You Get* model, or *WYNIWYG*. In other words, the system should configure itself automatically and load a *minimal* set of components required for executing the user applications in the most efficient way. The components are downloaded from the network, so only a small subset of system services are needed to bootstrap a node.

In the Automatic Configuration model, system and application software are composed of network-centric components, i.e., components available for download from a *Component Repository* present in the network. Component code is encapsulated in dynamically loadable libraries (DLLs in Windows and shared objects in Unix), which enables dynamic linking.

Each application, system, or component[2] specifies everything that is required for it to work properly (both hardware and software requirements). This collection of requirements is called *Prerequisite Specifications* or, simply, *Prerequisites*.

[2]From now on, we use the term "component" not only to refer to a piece of an application or system but also to refer to the entire application or system. This is consistent since, in our model, applications and systems are simply components that are made of smaller components.

3.1 Prerequisites

The prerequisites for a particular inert component (stored on a local disk or on a network component repository) must specify any special requirements for properly loading, configuring, and executing that component. We consider three different kinds of information that can be contained in a list of prerequisites.

1. The nature of the hardware resources the component needs.
2. The capacity of the hardware resources it needs.
3. The software services (i.e., other components) it requires.

The first two items are used by the Resource Management Service to determine where, how, and when to execute the component. QoS-aware systems can use these data to enable proper admission control, resource negotiation, and resource reservation. The last item determines which auxiliary components must be loaded and in which kind of software environment they will execute.

The first two items – reminiscent of the Job Control Languages of the mid-1960s – can be expressed by modern QoS specification languages such as QML [FK99b] and QoS aspect languages [LBS+98], or by using a simpler format such as SPDF (see Section 3.4.1). The third item is equivalent to the *require* clause in architectural description languages like Darwin [MDK94] and module interconnection languages like the one used in Polylith [Pur94].

The prerequisites are instrumental in implementing the WYNIWYG model as they let the system know what the exact requirements are, for instantiating the components properly. If the prerequisites are specified correctly, the system not only loads all the necessary components to activate the user environment, but also loads a minimal set of components required to achieve that.

We currently rely on the component programmer to specify component prerequisites. Mechanisms for automating the creation of prerequisite specifications and for verifying their correctness require further research and are beyond the scope of this paper. Another interesting topic for future research is

the refinement of prerequisites specifications at runtime according to what the system can learn from the execution of components in a certain environment. This can be achieved by using QoS profiling tools such as QualProbes [LN00].

3.2 Component Configurator

The explicit representation of *dynamic* dependencies is achieved through special objects attached to each relevant component at execution time. These objects are called *component configurators*; they are responsible for reifying the runtime dependencies for a certain component and for implementing policies to deal with events coming from other components.

While the Automatic Configuration Service parses the prerequisite specifications, fetches the required components from the Component Repository, and dynamically loads their code into the system runtime, it uses the information in the prerequisite specifications to create component configurators representing the runtime inter-component dependencies. Figure 2 depicts the dependencies that a component configurator reifies.

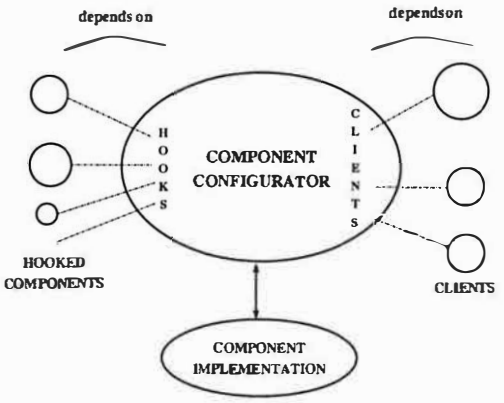


Figure 2: Reification of Component Dependencies

The dependencies of a component C are managed by a component configurator C^c. Each configurator C^c has a set of *hooks* to which other configurators can be attached. These are the configurators for the components on which C depends; they are called *hooked components*. The components that depend on C are called *clients*; C^c also keeps a list of references to the clients' configurators. In general, every time one defines that a component C_1 depends on a component C_2, the system should perform two actions:

1. attach C_2^c to one of the hooks in C_1^c and
2. add C_1^c to the list of clients in C_2^c.

Component configurators are also responsible for distributing events across the inter-dependent components. Examples of common events are the failure of a client and destruction, internal reconfiguration, or replacement of the implementation of a hooked component. The rationale is that such events affect all the dependent components. The component configurator is the place where programmers must insert the code to deal with these configuration-related events.

Component developers can program specialized versions of component configurators that are aware of the characteristics of specific components. These specialized configurators can, therefore, implement customized policies to deal with component dependencies in application-specific ways.

As an example of how customized component configurators could help applications, consider a QoS-sensitive video-on-demand client that reserves a portion of the local CPU for decoding a video stream. The application developer can program a special configurator that registers itself with the Resource Management Service. In this way, when the Resource Management Service detects a change in resource availability that would prevent the application from getting the desired level of service, it notifies the configurator (as shown in Figure 1). The configurator, with its customized knowledge about the application, sends a message to the video server requesting that the latter decrease the video frame rate. Then, with a lower frame rate, the client is able to process the video while the limited resource availability persists. When the resources go back to normal, another notification allows the video-on-demand configurator to re-establish the initial level of service.

3.3 Automatic Configuration Service

As described above, automatic configuration enables the construction of network-centric systems following a WYNIWYG model. To experiment with these ideas, we developed an Automatic Configuration Service for the *2K* operating system [KCM+00].

Different applications domains may have different ways of specifying the prerequisites of their application components. Therefore, rather than limiting the specification of prerequisites to a particular language, we built the Automatic Configuration Service as a framework in which different kinds of prerequisite descriptions can be utilized. To validate the framework, we designed the Simple Prerequisite Description Format (SPDF), a very simple, text-based format that allowed us to perform initial experiments. In the future, other more elaborated prerequisite formats including sophisticated QoS descriptions [FK99b, LBS+98] can be plugged into the framework easily.

In addition, depending upon the dynamic availability of resources and connectivity constraints, different algorithms for prerequisite resolution may be desired. For example, if a diskless PDA is connected to a network through a 2Mbps wireless connection, it will be beneficial to download all the required components from a central repository each time they are needed. On the other hand, if a laptop computer with a large disk connects to the network via modem, it will probably be better to cache the components in the local disk and re-use them whenever is possible.

Figure 3 shows how the architecture uses the two basic classes of the Automatic Configuration framework: *prerequisite parsers* and *prerequisite resolvers*. Administrators and developers can plug different concrete implementations of these classes to implement customized policies.

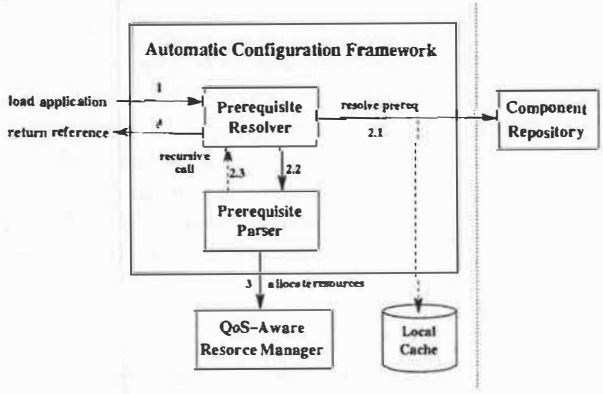


Figure 3: Automatic Configuration Framework

The automatic configuration process works as follows. First, the client sends a request for loading an application by passing, as parameters, the name of the application's "master" component and a reference to a component repository (step 1 in Figure 3). The request is received by the prerequisite resolver, which fetches the component code and prerequisite specification from the given repository, or from a local cache, depending on the policy being used (step 2.1).

Next, the prerequisite resolver calls the prerequisite parser to process the prerequisite specification (step 2.2). As it scans the specification, the parser issues recursive calls to the prerequisite resolver to load the components on which the component being processed depends (step 2.3). This may trigger several iterations over steps 2.1, 2.2, and 2.3.

After all the dependencies of a given component are resolved, the parser issues a call to the Resource Manager to negotiate the allocation of the required resources (step 3). After all the application components are loaded, the service returns a reference to the new application to the client (step 4).

3.4 A Concrete Implementation

To evaluate the framework, we created concrete implementations of the prerequisite parser and resolver. The prerequisite parser, called `SPDFParser`, processes SPDF specifications. The first prerequisite resolver, called `SimpleResolver`, uses CORBA to fetch components from the *2K* Component Repository. The second, called `CachingResolver`, is a subclass of `SimpleResolver` that caches the components on the local file system.

3.4.1 SPDF

We designed the Simple Prerequisite Description Format (SPDF) to serve as a proof-of-concept for our framework. An SPDF specification is divided in two parts, the first is called hardware requirements and the second, software requirements. Figure 4 shows an example of an SPDF specification for a hypothetical web browser. The first part specifies that this application was compiled for a Sparc machine running Solaris 2.7, that it requires at least 5MB of RAM memory but that it functions optimally with 40 MB of memory, and that it requires 10% of a CPU with speed higher than 300MHz.

The second part, software requirements, specifies

```
:hardware requirements
machine_type     SPARC
os_name          Solaris
os_version       2.7
min_ram          5MB
optimal_ram      40MB
cpu_speed        >300MHz
cpu_share        10%

:software requirements
FileSystem     CR:/sys/storage/DFS1.0 (optional)
TCPNetworking  CR:/sys/networking/BSD-sockets
WindowManager  CR:/sys/WinManagers/simpleWin
JVM            CR:/interp/Java/jvm1.2 (optional)
```

Figure 4: A Simple Prerequisite Description

that the web browser requires four components (or services): a file system (to use as a local cache for web pages), a TCP networking service (to fetch the web pages), a window manager (to display the pages), and a Java virtual machine (to interpret Java Applets).

The first line in the software requirements section specifies that the component that implements the file system (or the proxy that interacts with the file system) can be located in the directory `/sys/storage/DFS1.0` of the component repository (`CR`). It also states that the file system is an "optional" component, which means that the web browser can still function without a cache. Thus, if the Automatic Configuration Service is not able to load the file system component, it simply issues a warning message and continues its execution.

3.4.2 Simple Resolver and Caching Resolver

The `SimpleResolver` fetches the component implementations and component prerequisite specifications from the *2K* Component Repository. It stores the component code in the local file system and dynamically links the components to the system runtime. As new components are loaded, they are attached to hooks in the component configurator of the parent component, i.e., the component that required it. In the web browser example, the `SimpleResolver` would add hooks to the web browser configurator, call them `FileSystem`, `TCPNetworking`, `WindowManager`, and `JVM`, and attach the respective component configurators to each of these hooks.

Resolvers can be extended using inheritance. For example, with very little work, we extended the `SimpleResolver` to create a `CachingResolver` that checks for the existence of the component in the local disk (cache) before fetching it from the remote repository.

3.5 Simplifying Management

The Automatic Configuration Service simplifies management of user environments in distributed systems greatly. Whenever a new application is requested, the service downloads the most up-to-date version of its components from the network Component Repository and installs them locally. This provides several advantages including the following.

- It eliminates the need to upload components to the entire network each time a component is updated.
- It eliminates the need to keep track manually of which machines hold copies of each component because updates are automatic.
- It helps machines with limited resources, which no longer need to store all components locally.

3.6 Pushing Component Updates

The automatic configuration mechanism described here provides a pull-based approach for code updates and configuration. In other words, the service running in a certain network node takes the initiative to *pull* the code and configuration information from a Component Repository.

To support efficient and scalable management in large-scale systems, it may be desirable to allow system administrators to *push* code and configuration information into the network. Our architecture achieves this by using the concept of *mobile reconfiguration agents*, which we describe in detail elsewhere [KGA+00].

4 Resource Management Service

The Resource Management Service [Yam00] is organized as a collection of CORBA servers that are responsible for (1) maintaining information about the dynamic resource utilization in the distributed system, (2) locating the best candidate machine to execute a certain application or component based on its QoS prerequisites, and (3) allocating local resources for particular applications or components.

As shown in Figure 5, the Resource Management Service relies on Local Resource Managers (LRMs) present in each node of the distributed system. The LRM's task is to export the hardware resources of a particular node to the whole network. The distributed system is divided in clusters and each cluster is managed by a Global Resource Manager (GRM).

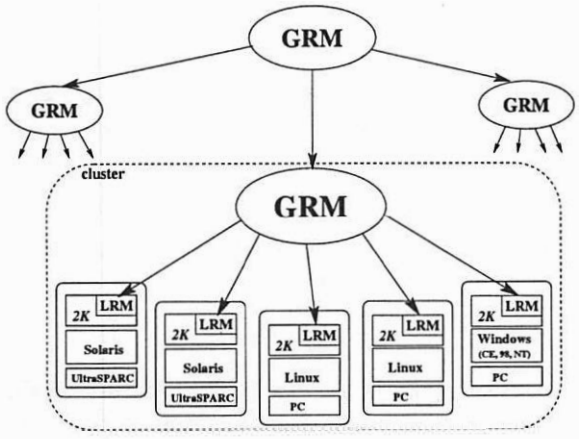


Figure 5: Resource Management Service

4.1 Resource Monitoring

The LRMs running in each network node send updates of the state of their resources (e.g., CPU and memory usage) to the GRM periodically. The GRM implementation encompasses an instance of the standard OMG Object Trading Service [OMG98]. A reference to the LRM of each machine in the cluster is stored in the GRM database as a trader "service offer" and the state of its resources is stored as the offer's "properties".

To reduce network and GRM load, it is important to limit the frequency in which the LRMs send their updates to the GRM. Thus, although LRMs check the state of their local resources frequently (e.g., every ten seconds), they only send this information to the GRM when (1) there were significant changes in resource utilization since the last update (e.g., a variation in more than 20% on the CPU load) or (2) a certain time has passed since the last update was sent (e.g., three minutes). In addition, when a machine leaves the network, in case of a shutdown or a voluntary disconnection of a mobile computer, the LRM unregisters itself from the GRM database. If the GRM does not receive an update from an LRM for a period twice as long as the time in item 2 above, it assumes that the machine with that LRM is unaccessible.

4.2 Resource Reservation

The LRMs are also responsible for performing QoS-aware admission control, resource negotiation, reservation, and scheduling of tasks on a single node. This is achieved with the help of a Dynamic Soft Real-Time Scheduler [NhCN98] that runs as a user-level process in conventional operating systems like Solaris and Windows. The LRM works as a CORBA wrapper for this scheduler, which uses the system's low-level real-time API to provide QoS guarantees to applications with soft real-time requirements.

This CORBArized scheduler can be used at any time by CORBA clients to request QoS guarantees on the availability of CPU and memory. For example, as explained in Section 3.3, a prerequisite parser may issue requests to reserve CPU and memory based on a component's hardware prerequisite specifications.

4.3 Executing Applications

Both the LRM and the GRM export an interface that let clients execute applications (or components) in the distributed system. The GRM maintains an approximate view of the cluster resource utilization state and it uses this information as a hint for performing QoS-aware load distribution within its cluster.

When a client wishes to execute a new application, it sends an `execute_application` request to the local LRM. The LRM checks whether the local machine has enough resources to execute the application comfortably. If not, it forwards the request to

the GRM. The latter uses its information about the resource utilization in the distributed system to select a machine that would be the best candidate to execute that application and forwards the request, as a oneway message, to the LRM of that machine. The LRM of the latter machine tries to allocate the resources locally, if it is successful, it sends a oneway ACK message to the client LRM. If it is not possible to allocate the resources on that machine, it sends a NACK back to the GRM, which then looks for another candidate machine. If the GRM exhausts all the possibilities, it returns an empty offer to the client LRM.

When the system finally locates a machine with the proper resources, it creates a new process to host the application. Next, it uses the Automatic Configuration Service to fetch all the necessary components (i.e. the master component's dependencies) from the Component Repository and dynamically load them into that process as described in Section 3.3.

4.3.1 Client Request Format

The format of the client request to the initial LRM is the following.

```
CosTrading::OfferSeq execute_application (
   in string categoryName,
   in string componentName,
   in string args,
   in CosTrading::PropertySeq QoS_spec,
   in CosTrading::Constraint platform_spec,
   in CosTrading::Preference prefs,
   in CosTrading::Lookup::SpecifiedProps
                                return_props
);
```

categoryName/componentName specify which of the components in the *2K* Component Repository is the master component of the application to be executed and args contains the arguments that should be passed to it at startup time.

QoS_spec defines the quality of service required for this application. It is specified as a list of <resourceName,resourceValue> pairs. As an example, if the resource is the CPU, then the resource value should be a structure of the following format (specified by the scheduler's CPU server [NhCN98]).

```
struct CpuReserve {
   long serviceClass;
   long period;
   long peakProcessingTime;
   long sustainableProcessingTime;
   long burstTolerance;
   float peakProcessingUtil;
};
```

platform_spec is the criteria to select a cluster node and it is specified using the OMG Trader Constraint Language. For example, (os_name == 'Linux') and (processor_util < 40) will select a Linux machine whose CPU utilization is less than 40%.

prefs specifies the preferred machine in case multiple machines satisfy the requirements. For example, max(RAM_free) will select the machine with the maximum available physical memory.

Finally, return_props specifies which properties (resource utilization information) should be included in the service offer that is returned. The returned value also includes a reference to the component configurator (see Section 3.2) of the new application.

4.4 Fault-Tolerance and Scalability

To provide fault-tolerance and scalability, the Resource Management Service architecture depends on a collection of replicated GRMs in each cluster. LRMs send their updates as a multicast message to all the GRMs in the cluster. Since, strong consistency between the GRMs is not required, we can use an unreliable multicast mechanism. Client requests are sent to a single GRM and different clients may use different GRMs for load balancing.

To enhance scalability across multiple clusters connected through the Internet, GRMs can be federated in a hierarchical way. If a request cannot be resolved in a particular cluster, the GRM forwards it to a parent GRM in the hierarchy. The parent GRM maintains an approximate view of the resource utilization in its child clusters and uses this information as a hint to locate a proper cluster to fulfill the client request.

Although we have designed the protocols and algorithms for fault-tolerance and scalability mentioned

in this subsection, their implementation is still underway.

5 Implementation and Experimental Results

The Automatic Configuration Service is implemented as a library that can be linked to any application. A program enhanced with this service becomes capable of fetching components from a remote Component Repository and dynamically loading and assembling them into its local address-space. The library requires only 157Kbytes of memory on Solaris 7, which makes it possible to use it even on machines with limited resources such as a PalmPilot. In fact, we expect that services similar to this will be extensively used in future mobile systems to configure software automatically according to location and user requirements.

To evaluate the performance of the Automatic Configuration Service, we instrumented a test application [KCN00] to measure the time for fetching, dynamic linking, and configuring its constituent components.

5.1 Loading Multiple Components

Figure 6 shows the total time for the service to load from one to eight components of 19.2Kbytes each. These experiments were carried out on two Sparc Ultra-60 machines running Solaris 7 and connected by a 100Mbps Fast Ethernet network. The Component Repository was executed on one of the machines and the test application with the Automatic Configuration Service on the other. Each value is the arithmetic mean of five runs of the experiment. The vertical bars in the subsequent graphs and the numbers in parentheses in Table 1 represent the standard deviation. As the graph shows, the variation in execution times across different runs of the experiment was very small.

Table 1 shows, in more detail, how the service spends its time when loading a single 19.2Kbyte component. The current version of the Automatic Configuration Service fetches the prerequisites file from the remote Component Repository and saves it to the local disk. The same is done with the file

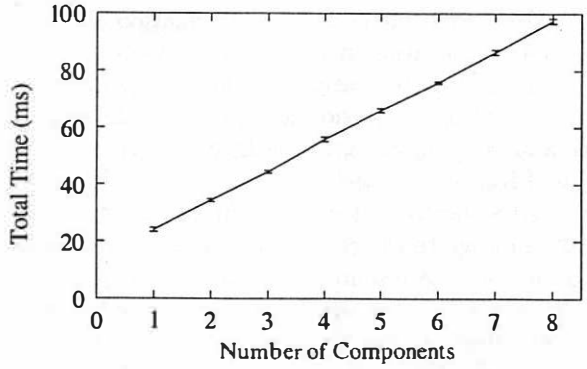


Figure 6: Automatic Configuration Service Performance

containing the component code. Then, it uses the underlying operating system to perform the local dynamic linking of the component into the process runtime.

The table also shows the additional time spent by the service (row labeled as "autoconf protocol additional operations") to detect if there are more components or prerequisite files to load, to parse the prerequisite file, and to reify dependencies. This overhead accounts for 46% of the total time required to load the component, which suggests that it would be desirable to improve this part of the service by optimizing the implementation of the *SimpleResolver* (see Section 3.4). We believe that an optimized version of the *SimpleResolver* could lead to improvements in the order of 20% for components of this size.

In the experiments described in this section, the component code and prerequisite files were cached in the memory of the machine executing the Component Repository. When the Component Repository program needs to read both files from its local disk, there is an additional overhead of approximately 20 milliseconds.

5.2 Components of Different Sizes

To evaluate how the time for loading a single component varies with the component size, we created a program that generates components of different sizes. According to its command-line arguments, this program generates C++ source code containing a given number of functions (which include code to perform simple arithmetic operations) and local and

Action	Time (ms)	% of the total
fetching prerequisites from Component Repository	2 (0)	8
saving prerequisites to local disk	1 (0)	4
fetching component from Component Repository	4 (0)	17
saving component to local disk	1 (0)	4
local dynamic linking	5 (0)	21
autoconf protocol additional operations	11 (0.7)	46
Total	24 (0.7)	100

Table 1: Discriminated Times for Loading a 19.2Kbyte Component

global variables. Using this program, we created components whose DLL sizes vary from 12 to 115 Kbytes. Figure 7 shows the time for the Automatic Configuration Service to load a single component as the component size increases.

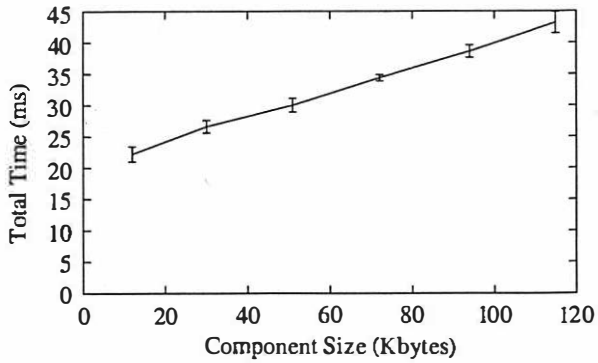


Figure 7: Times for Loading Components of Different Sizes

Figure 8 shows the absolute times spent in each step of the process[3]. We can notice that the time spent in the item labeled "autoconf protocol" is approximately constant[4]. Hence, as the component size increases, its relative contribution to the total time decreases. This can be noticed in Figure 9, which shows the same data in a different form. In this case, the figure shows the percentage of the total time spent in each of the steps of the process.

As the size of the component increases, the time for fetching the code from the remote repository to the local machine becomes the dominant factor. It is important to remember that these data were captured in a fast local network. If the access to the repository requires the use of a lower bandwidth connection, then this step would clearly be the most

[3]These steps are the same as those presented in Table 1.

[4]This is expected since the messages processed in this step do not carry component code and therefore are not affected by the size of the component.

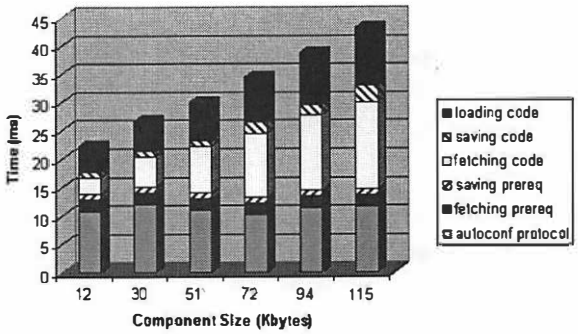


Figure 8: Discriminated Times for Loading Components of Different Sizes

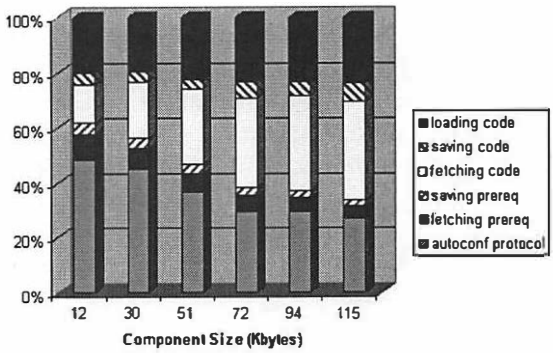


Figure 9: Discriminated Percentual Times for Loading Components of Different Sizes

important with respect to performance. This suggests that deriving intelligent algorithms for component caching, taking component versions and user access patterns into consideration is an important topic for future research.

Although there is still much room for improvements and performance optimizations in the protocols used by the Automatic Configuration Service, the results presented here are very encouraging. They demonstrate that it is possible to carry out automatic con-

figuration of a distributed component-based application within a tenth of a second, which is what we intended to prove.

5.3 Resource Management

We have not yet carried out an extensive performance evaluation of the Resource Management Service. However, preliminary results [Yam00] show that the overhead imposed by the LRMs in the individual nodes is low and that the time for launching a simple remote application through the GRM and LRM is in the order of a tenth of a second. As future work, we intend to carry out a comprehensive evaluation of this service.

6 Related Work

The OMG CORBA Component Model (CCM) specifies a standard framework for building, packaging, and deploying CORBA components [OMG99]. Unlike our model, which focuses on prerequisites and dynamic dependencies, the CORBA Component Model concentrates on defining an XML vocabulary and an extension to the OMG IDL to support the specification of component packaging, customization, and configuration. The CCM *Software Package Descriptor* is reminiscent of our SPDF as it contains a description of package dependencies, i.e., a list of other packages or implementations that must be installed in the system for a certain package to work. *CORBA Component Descriptors*, on the other hand, describe the interfaces and event ports used and provided by a CORBA component.

We believe that our model and CCM complement each other and could be integrated. CCM provides a static description of component needs and interactions, while our model manages the runtime dynamics. Although CCM was already approved by OMG, publicly available ORBs do not support it yet. Once this happens, we intend to work towards the integration of the two models.

Among the major CORBA implementations, the one that most resembles our work is Orbix 2000 [ION00]. Its *Adaptive Runtime Architecture* lets users add functionality to the ORB by loading plug-ins dynamically. Whenever a request is sent to the ORB, it is processed by a chain of interceptors that can be configured in different ways using the loaded plug-ins. In that way, the ORB can be configured with interceptors that implement security, transactions, different transport protocols, etc. When the ORB loads a plug-in, it checks its version and dependence information. A centralized configuration repository specifies plug-in availability and configuration settings. Using this architecture it could be relatively easy to implement the functionality provided by our Automatic Configuration and Resource Management Services.

Enterprise JavaBeans [Tho98] is a server-side technology for the development of component-based systems. It does not support the functionality for Automatic Configuration and Resource Management provided in our system. Nevertheless, it provides *deployment descriptors* that let one define, at deployment time, the configuration of individual components (Beans). Instead of recording the configuration information in a text format – like our SPDF and CORBA's XML formats – deployment descriptors are serialized Java classes. A deployment descriptor can customize the behavior of a Bean by setting environment properties as well as define runtime attributes of its execution context, such as security, transactions, persistence, etc. [MH00].

Jini is a set of mechanisms for managing dynamic environments based on Java. It provides protocols to allow services to *join* a network and *discover* what services are available in this network. It also defines standard service interfaces for leasing, transactions, and events [AOS+99]. When a Jini server registers itself with the Jini lookup service, it stores a piece of Java byte code, called proxy, in its entry in the lookup service. When a Jini-enabled client uses the lookup service to locate the server, it receives, as a reply, a *ServiceItem*, which is composed of a service ID, the code for the proxy, and a set of service attributes. The proxy is then linked into the client address-space and is responsible for communication with the server. In this way, the communication between the client and the server can be customized, and optimized protocols can be adopted.

This Jini mechanism for proxy distribution can be achieved in a CORBA environment by using the Automatic Configuration Service in conjunction with a reflective ORB such as *dynamicTAO* [KRL+00]. The Automatic Configuration Service would fetch the proxy code and dynamically link it, while *dynamicTAO* would use the TAO pluggable protocols

framework [OKS+00] to plug the proxy code into the TAO framework.

Jini is normally limited to small-scale networks and it does not address the management of component-based applications and inter-component dependence. Due to the large memory requirements imposed by Java/Jini, this is not yet a viable alternative for most PDAs and embedded devices.

The Globus project [FK98] provides a "computational grid" [FK99a] integrating heterogeneous distributed resources in a single wide-area system. It supports scalable resource management based on a hierarchy of resource managers similar to the ones we propose. Globus defines an extensible Resource Specification Language (RSL) that is similar to our SPDF (described in Section 3.4.1). RSL [Glo00] allows Globus users to specify the executables they want to run as well as their resource requirements and environment characteristics. RSL could be integrated in our system by plugging an *RSLParser* into our Automatic Configuration framework. A fundamental difference between Globus and our work is that we focus on *component-based* applications that are dynamically configured by assembling components fetched from a network repository. In Globus, on the other hand, the user specifies the application to be executed by giving the name of a single executable on the target host file system or by giving a URL from which the executable can be fetched.

Legion [GW+97] is the system that shares most similarities with *2K* as it also builds on a distributed, reflective object model. However, the Legion researchers focused on developing a new object model from scratch. Legion applications must be built using Legion-specific libraries, compiler, and run-time system (the Legion's ORB). In contrast, we focused on leveraging CORBA technology to build an integrated architecture that could provide the same functionality as Legion, while still preserving complete interoperability with other CORBA systems. In addition, our work emphasizes automatic configuration and dependence management, which are not addressed by Legion.

Systems based on architectural connectors like UniCon [SDZ96] and ArchStudio [OT98] and systems based on software buses like Polylith [Pur94] separate issues concerning component functional behavior from component interaction. Our model goes one step further by separating inter-component communication from inter-component dependence. Connectors and software buses require that applications be programmed to a particular communication paradigm. Unlike previous work in this area, our model does not dictate a particular communication paradigm like connectors or buses. It can be used in conjunction with connectors, buses, local method invocation, CORBA, Java RMI, and other methods. As demonstrated by our experiments with *dynamicTAO* [KRL+00], the model was applied to a legacy system without requiring any modification to its functional implementation or to its inter-component communication mechanisms.

Communication and dependence are often intimately related. But, in many cases, the distinction between inter-component dependence and inter-component communication is beneficial. For example, the quality of service provided by a multimedia application is greatly influenced by the mechanisms utilized by underlying services such as virtual memory, scheduling, and memory allocation (e.g., through the `new` operator). The interaction between the application and these services is often implicit, i.e., no direct communication (e.g., library or system calls) takes place. Yet, if the system infrastructure allows developers to establish and manipulate dependence relationships between the application and these services, the application can be notified of substantial changes in the state and configuration of the services that may affect its performance.

Research in software architecture [SG96] and dynamic configuration [PCS98] typically focuses on the architecture of individual applications. It does not deal with dependencies of application components towards system components, other applications, or services available in the distributed environment. Our approach differs from them in the sense that, for each component, we specify its dependencies on all the different kinds of environment components and we maintain and use these dynamic dependencies at runtime. Approaches based on software architecture typically rely on global, centralized knowledge of application architecture. In contrast, our method is more decentralized and focuses on more direct component dependencies. We believe that, rather than conflicting with the software architecture approach, our vision complements them by reasoning about *all* the dependencies that may affect reliability, performance, and quality of service.

The final solution to the problem of supporting reliable automatic (re)configuration may reside on the

combination of our model with recent work in software architecture and dynamic (re)configuration. This is certainly an important open research problem to be investigated in the future.

7 Future Work

Under the *2K* project we have also been working on QoS compilation techniques, addressing the problem of translating application-level QoS specifications to component-level QoS specifications, and then to resource-level QoS specifications [NWX00]. In the near future, our group will work on the implementation of the mechanisms for fault-tolerance and scalability described in Section 4.4. Security will be provided by a CORBA implementation of the standard Generic Security Services (GSS) API [Lin97].

In the previous sections, we alluded to some other important topics for future work, namely, (1) automatic creation and refinement of prerequisite specifications, (2) intelligent algorithms for component caching taking versions into consideration, and (3) the integration of our dependence model with recent research in software architecture.

8 Conclusions

Component technologies will play a fundamental role in the next generation computer systems as the complexity of software and the diversity and pervasiveness of computing devices increase. However, component technologies must offer mechanisms for automatic management of inter-component dependencies and component-to-resource dependencies. Otherwise, the development of component-based systems will continue to be difficult and frequently lead to unreliable and non-robust systems.

Although there are still a number of open problems for future research, we believe that this paper gives an important contribution to the area by presenting an object-oriented architecture for automatic configuration and dynamic resource management in distributed component systems. Performance evaluation demonstrated that our system is able to dynamically instantiate applications by assembling network components in less than a tenth of a second.

Future work in our group will extend the Resource Management Service implementation to improve its fault-tolerance and scalability and enhance the synergy between dynamic resource management and automatic configuration.

Acknowledgments The authors gratefully acknowledge the wise input provided by Klara Nahrstedt, Francisco Ballesteros, Manuel Román, Dulcineia Carvalho, Dilma Silva, and the *2K* team. We thank the anonymous reviewers for their useful comments, which helped to improve the quality of this paper.

Availability Source code and more information about the Automatic Configuration Service and the Resource Management Service can be found at `http://choices.cs.uiuc.edu/2k`.

References

[AOS+99] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification.* Addison-Wesley, June 1999.

[CKB+00] Dulcineia Carvalho, Fabio Kon, Francisco Ballesteros, Manuel Román, Roy Campbell, and Dennis Mickunas. Management of Execution Environments in 2K. In *Proceedings of the Seventh International Conference on Parallel and Distributed Systems (ICPADS'2000)*, pages 479–485. IEEE Computer Society, July 2000.

[FK98] I. Foster and C. Kesselman. The Globus Project: A Status Report. In *Proceedings of the IPPS/SPDP '98 Heterogeneous Computing Workshop*, pages 4–18, 1998.

[FK99a] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, San Francisco, 1999.

[FK99b] Svend Frølund and Jari Koistinen. Quality of Service Aware Distributed Object Systems. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technology and Systems (COOTS'99)*, pages 69–83, San Diego, May 1999.

[Glo00] The Globus Project. *Globus Resource Specification Language RSL v1.0*, 2000. Available at `http://www.globus.org/gram`.

[GW+97] Andrew S. Grimshaw, Wm. A. Wulf, et al. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1), January 1997.

[ION00] IONA Technologies. *Orbix 2000*, March 2000. White paper available at `http://www.iona.com`.

[KC99] Fabio Kon and Roy H. Campbell. Supporting Automatic Configuration of Component-Based Distributed Systems. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, pages 175–187, San Diego, CA, May 1999.

[KCM+00] Fabio Kon, Roy H. Campbell, M. Dennis Mickunas, Klara Nahrstedt, and Francisco J. Ballesteros. 2K: A Distributed Operating System for Dynamic Heterogeneous Environments. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC'9)*, pages 201–208, Pittsburgh, August 2000.

[KCN00] Fabio Kon, Roy H. Campbell, and Klara Nahrstedt. Using Dynamic Configuration to Manage A Scalable Multimedia Distribution System. *Computer Communication Journal (Special Issue on QoS-Sensitive Distributed Systems and Applications)*, Fall 2000. Elsevier Science Publisher.

[KGA+00] Fabio Kon, Binny Gill, Manish Anand, Roy H. Campbell, and M. Dennis Mickunas. Secure Dynamic Reconfiguration of Scalable CORBA Systems with Mobile Agents. In *Proceedings of the IEEE Joint Symposium on Agent Systems and Applications / Mobile Agents (ASA/MA'2000)*, pages 86–98, Zurich, September 2000.

[KRL+00] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in LNCS, pages 121–143, New York, April 2000. Springer-Verlag.

[LBS+98] J. P. Loyall, D. E. Bakken, R. E. Schantz, J. A. Zinky, D. A. Karr, R. Vanegas, and K. R. Anderson. QoS Aspect Languages and Their Runtime Integration. In *Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, Pittsburgh, Pennsylvania, May 1998.

[Lin97] J. Linn. The Generic Security Service Application Program Interface (GSS API). Technical Report Internet RFC 2078, Network Working Group, January 1997.

[LN00] Baochun Li and Klara Nahrstedt. QualProbes: Middleware QoS Profiling Services for Configuring Adaptive Applications. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in LNCS, pages 256–272, New York, April 2000. Springer-Verlag.

[MDK94] Jeff Magee, Naranker Dulay, and Jeff Kramer. Regis: A Constructive Development Environment for Distributed Programs. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 1(1):37–47, 1994.

[MH00] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, 2nd edition, March 2000.

[NhCN98] Klara Nahrstedt, Hao hua Chu, and Srinivas Narayan. QoS-aware Resource Management for Distributed Multimedia Applications. *Journal of High-Speed Networking, Special Issue on Multimedia Networking*, 7:227–255, 1998.

[NWX00] Klara Nahrstedt, Duangdao Wichadakul, and Dongyan Xu. Distributed QoS Com-

pilation and Runtime Instantiation. In *Proceedings of the IEEE/IFIP International Workshop on QoS (IWQoS'2000)*, Pittsburgh, June 2000.

[OKS+00] C. O'Ryan, F. Kuhns, D. C. Schmidt, O. Othman, and J. Parsons. The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, New York, April 2000.

[OMG98] OMG. *CORBAservices: Common Object Services Specification.* Object Management Group, Framingham, MA, 1998. OMG Document 98-12-09.

[OMG99] OMG. *CORBA Components.* Object Management Group, Framingham, MA, 1999. OMG Document orbos/99-07-01.

[OT98] Peyman Oreizy and Richard N. Taylor. On the Role of Software Architectures in Runtime System Reconfiguration. In *Proceedings of the 4th International Conference on Configurable Distributed Systems (CDS'98)*, Annapolis, Maryland, USA, May 1998.

[PCS98] Jim Purtilo, Robert Cole, and Rick Schlichting, editors. *Fourth International Conference on Configurable Distributed Systems.* IEEE, May 1998.

[Pur94] James Purtilo. The Polylith Software Bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, January 1994.

[SDZ96] Mary Shaw, R. DeLine, and G. Zelesnik. Abstractions and Implementations for Architectural Connections. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems (CDS'96)*, Annapolis, Maryland, USA, May 1996.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[Tho98] Anne Thomas. *Enterprise JavaBeans Technology: Server Component Model for the Java Platform.* Patricia Seybold Group, December 1998. Available at `http://java.sun.com/products/ejb/white`.

[Wei92] Mark Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):94–104, September 1992.

[Yam00] Tomonori Yamane. The Design and Implementation of the 2K Resource Management Service. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, February 2000.

An Adaptive Data Object Service for Pervasive Computing Environments*

Christopher K. Hess[1] Francisco Ballesteros[2] Roy H. Campbell[1]
M. Dennis Mickunas[1]

ckhess@cs.uiuc.edu, nemo@gsyc.escet.urjc.es, {roy, mickunas}@cs.uiuc.edu

[1]Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

[2]Rey Juan Carlos University of Madrid
Tulipan s/n. Mostoles
Madrid, Spain

Abstract

Workstations and PCs typically are rich in resources, in contrast to palmtop devices, which are generally quite limited. This disparity offers challenges to integrating these heterogeneous devices into a single distributed system. Services must be available to each device, but it may be necessary to modify certain services if the connected device does not have the desired resources.

A key component of many distributed systems is remote access to data. Traditional distributed file systems are typically rather static and are not able to adapt to the current available resources of the devices involved. Data files are treated as continuous streams of bytes and the interfaces to access them are designed for unstructured data; they simply transfer buffers of contiguous data. Providing modality and adapting content using these interfaces proves difficult.

In this paper, we present an adaptive data object service for pervasive computing environments using distributed objects. Data is manipulated through an object-oriented interface based on containers and iterators. The interface is also used to model data operations, conversions, and proxies. The system is aware of its environment and can instantiate objects in the proper locations to optimize performance.

*This research is supported by a grant from the National Science Foundation, NSF 98-70736.

1 Introduction

The recent popularity of personal digital assistants (PDAs) and Web-enabled cell phones has brought mobile handheld computing into the mainstream. Users are now able to perform many tasks that were once restricted to larger desktop systems. Although these devices will almost certainly always possess less computing power than their desktop counterparts, they will eventually offer universal access to the network. One of the key challenges is the integration of these handheld devices into larger distributed systems. The handheld devices should become an extension of the system that they can interact with in the same way that a stationary machine can.

The increasing diversity of devices accessing distributed systems makes traditional data distribution mechanisms inappropriate, since differing device types may require the service to behave in different ways. For example, when displaying video on a small device, it may be better to decode MPEG on a nearby host and send raw pixmaps to the handheld used for output. Systems that are not able to adapt to the current environment are therefore not best suited for heterogeneous distributed systems.

An active area of research involving highly heterogeneous environments has been that of pervasive computing [Wei93, Abo99, MIT, Hew, Mic]. These environments consist of intelligent rooms or areas, containing appliances (whiteboard, video pro-

jectors, etc), powerful stationary computers, and mobile wireless handheld devices. The large collection of devices, resources, and peripherals must be coordinated and access to them must be made simple. Such coordination may be viewed as being analogous to the role of a traditional operating system. However, the heterogeneity, mobility, and sheer number of devices makes the system vastly more complex [RC00]. Applications may have the choice of a number of input devices, such as mouse, pen, or finger; output devices, such as monitor, PDA screen, wall-mounted display, or speakers. An infrastructure for such a space must be able to locate the most appropriate device, detect when new devices are spontaneously added to the system, and adapt content when data formats are not compatible with output devices. For example, if a user wishes to view an on-going presentation on a small handheld, images of the slides could be sent to the roaming user, but in a format more appropriate for the device, such as a scaled down image to fit the small screen size. Moreover, more extreme transformations may be performed, such as converting text data to audio. Applications should not be bothered with the complexities of such conversions; they should gain access to data in a particular format by simply opening the data source as the specific desired type. The system should automatically adapt content to the desired format and place the conversion modules in locations to maximize efficiency.

To address the foregoing issues, we have built a general data distribution service targeted at heterogeneous environments, that incorporates automatic content adaptation, location awareness, and knowledge of environment. The design of the service is based on the concept of containers and iterators exhibited in the Standard Template Library (STL) [SL94, MS96]; containers provide data manipulation operations, parsing mechanisms, and content transformations for structured data and convenient access is provided via iterators. Containers may be instantiated in the most appropriate locations, and access to these components may be transfered among nodes, enabling containers placed on various nodes to communicate. The application programming interface uses C++ templates and generic programming [Mus89] concepts to hide the communication infrastructure and maximize code reuse. In the current implementation, we have used CORBA as the underlying middleware layer. However, we are not restricted to using CORBA and are planning on porting the system to a light-weight communication core.

The remainder of this paper is presented as follows: section 2 gives an overview of our data service, including a brief description of the larger system the service is a part of. Section 3 describes the system layer of the service and the user layer containers and iterators, including examples. Section 4 presents our continuing work. Sections 5 and 6 present related work and concluding remarks, respectively.

2 The Data Object Service

The Data Object Service (*DOS*) is the data delivery mechanism for *Gaia*, an operating system for physical spaces we are currently developing. In the following sections, we describe *Gaia* and the design of the data service.

2.1 Overview of Gaia

Gaia is an infrastructure that exports and coordinates the resources contained in a physical space, thereby defining a generic computational environment [Gai00]. Gaia converts physical spaces and the ubiquitous computing devices they contain into a programmable computing system. *Gaia* is analogous to traditional computing systems; just as a computer is viewed as one object, composed of input/output devices, resources and peripherals, so is a physical space populated with many devices. An operating system for such a space must be able to coordinate the resources available in such a space. *Gaia* is similar to traditional operating systems by managing the tasks common to all applications built for physical spaces.

Gaia provides some core services, including events, entity presence (devices, users, and services), discovery, naming, location, trading. Devices are able to detect when they have entered new spaces and can take advantage of the services available in the physical location. By specifying well-defined interfaces to devices and services, applications may be built in a generic way that are able to run in arbitrary spaces. For example, a classroom application may be built that uses the physical devices in a room. When the user moves to a new classroom, the application can use the devices present in the new space.

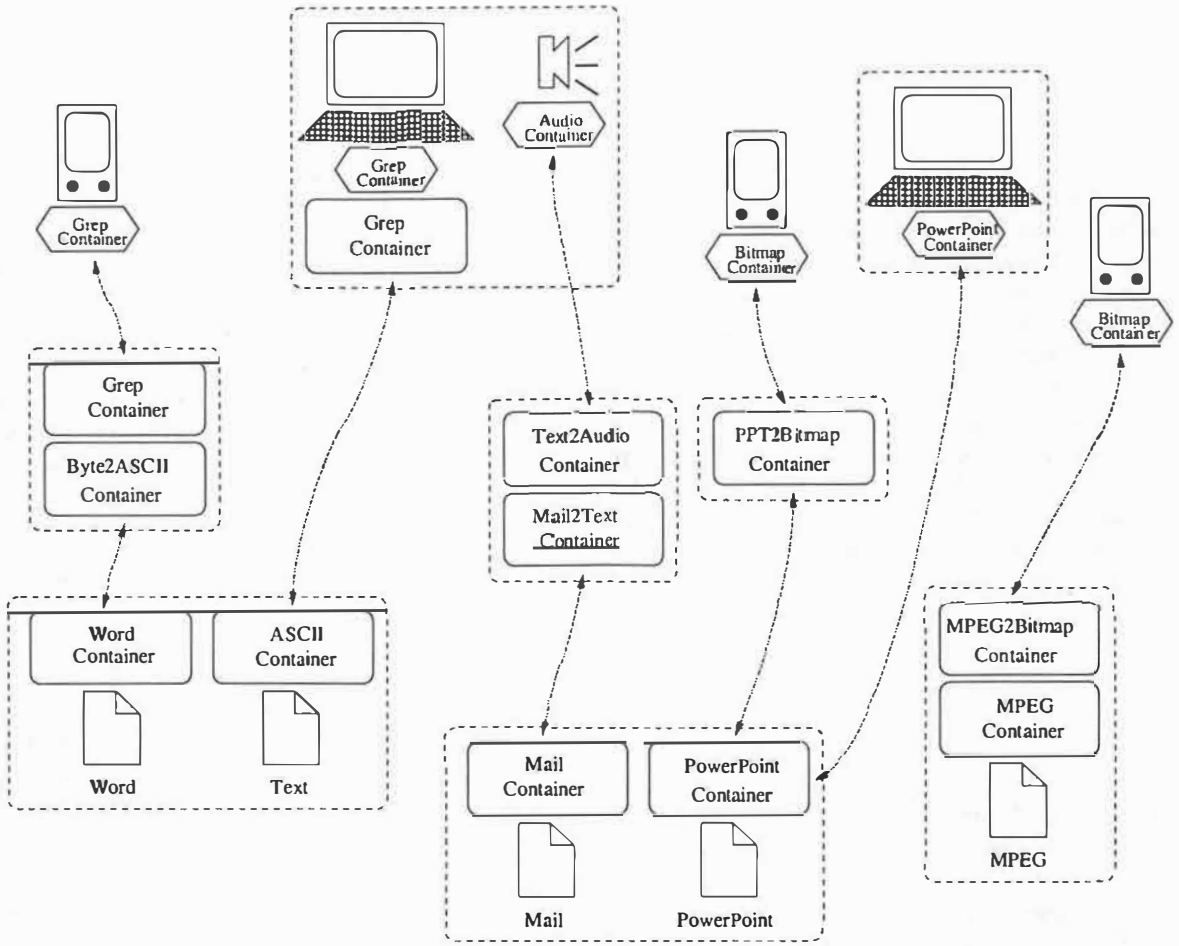


Figure 1: Servers manage their local native files and devices. The system can instantiate containers on any node in the system and adapt content for different device types. Rounded rectangles represent container instances. Hexagons represent template wrappers.

In addition, we are developing an application model for pervasive computing environments that is inspired from the Model-View-Controller (MVC) architecture. The MVC components are no longer restricted to software entities, i.e., they may be physical entities. For example, a house may be the model, providing data to various views of its sub-systems. Our model introduces an *adaptor* that can modify data from the model to a format the view desires.

Gaia is an extension of our previous work on the *2K* operating system. *2K* is a middleware operating system using CORBA [The98] as the communication mechanism and runs on top of existing platforms, such as Windows NT and Solaris. It uses a modified version of the TAO Object Request Broker (ORB) [SC99], called *dynamicTAO* [KRL+00], that offers dynamic configuration of the ORB internal engine in order to adapt to the dynamic needs of users.

Our initial implementation of *DOS* uses CORBA to leverage some of the standard CORBA services. In *Gaia*, we are applying these services in the context of physical spaces. Two services that are used heavily are the Name Service and the Trading Service. The Name Service allows transparent access to particular object references for applications. The Trading Service allows applications to find objects that possess some specific constraints. We are using these services in conjunction to provide location specific trading services. The Event Service is used with physical location detection systems (e.g., badges) to provide presence notifications of entities, such as users and devices. In addition, events are used to send "heartbeat" messages to the system to

determine entity liveness.

Since *Gaia* targets pervasive computing environments, many small devices interact with the system. In the future, we will use a small composable communication mechanism, called the Universal Interoperable Core (UIC), that can communicate via different protocols (e.g., GIOP, SOAP) for mobile handheld devices that users may carry [UBI00]. The UIC can be composed dynamically, using only the required components. This allows the implementation to be customized to small devices and allows these devices to interact with services using standard protocols. In addition, devices can include server-side functionality, allowing them to accept events and method invocations. Since UIC is able to communicate with standard CORBA servers, we will be able to access the standard and custom services from these small handheld devices.

2.2 Data Objects in Gaia

Traditional distributed file systems [How88, SGK+85, Wel92] are generally designed for homogeneous environments and simply transfer data to the local node. However, the heterogeneous nature of pervasive computing environments deems the static configurations of traditional distributed file systems inappropriate, since some nodes (e.g., handhelds) may require additional support from the infrastructure. Fixed policies may preclude some nodes from participating in these environments. A data access service that is dynamically configurable offers modality for different device types.

DOS is a middleware data service that makes use of the native operating system to manage data on disk. However, the service offers more than simple access to file data. In general, data is no longer transported as streams of bytes (although this mode is supported), but as data objects. Traditional file system interfaces (i.e., *open*, *read*, *write*, *close*) are replaced with object-oriented abstractions: *containers* and *iterators* [GHJV95, SL94]. These abstractions are a more suitable interface for accessing data as objects, since iterators can return data of a certain type and can be used to traverse the objects. Iterators provide the indirection needed to manipulate different containers using a single interface. In contrast to the standard *read* method for example, which passes a buffer to be filled in, an iterator returns references to objects whose size may be unknown *a priori* to the user.

In the most basic form, containers are simply wrappers for native file data or directories, but they can also be much more interesting and useful objects. In general, containers may represent *any* collection of data, that may be generated on-the-fly, gathered from disparate sources, or common data shared among distributed applications. They may also be used to interface with devices (e.g., writing a postscript file to a printer).

Containers are constructed as CORBA objects and applications can communicate with them through ORBs. CORBA provides infrastructure for transparent and platform-independent access to remote (or local) objects. Objects can be instantiated on any host and references to these objects can be passed around in a simple manner. This facilitates the creation of containers in various locations that may easily be connected together, as illustrated in Fig. 1. Dynamic placement of objects (and their functionality) is critical for heterogeneous environments to support all device capabilities. Different containers hold different kinds of data and CORBA handles the job of marshaling/unmarshaling and transporting the data. *DOS* assumes some of the burden generally placed on the programmer by parsing native file contents into indexed components that applications can manipulate more easily.

Some containers may be instantiated on *proxy servers*. These servers generally do not provide clients access to disk, but rather to their CPU and memory. For example, a proxy [Sha86] may be used to perform some expensive parsing or computation that should not be performed on the node maintaining the native files (as not to hinder other clients interacting with that server) or on the client (it is too weak to perform the parsing itself). The system can configure itself by placing container objects in the best location, based on knowledge of surrounding devices, to optimize performance.

For example, when performing a *grep* on a collection of files, simply copying data as-is to a small handheld device and searching locally may be inappropriate due to the severe resource limitations. It may be better to find matches on the file server and then transfer the resulting text to the handheld, as illustrated in Fig. 2. Clients can then use the search results to retrieve only those files that are of interest. However, the system should be configurable to direct where such operations should be

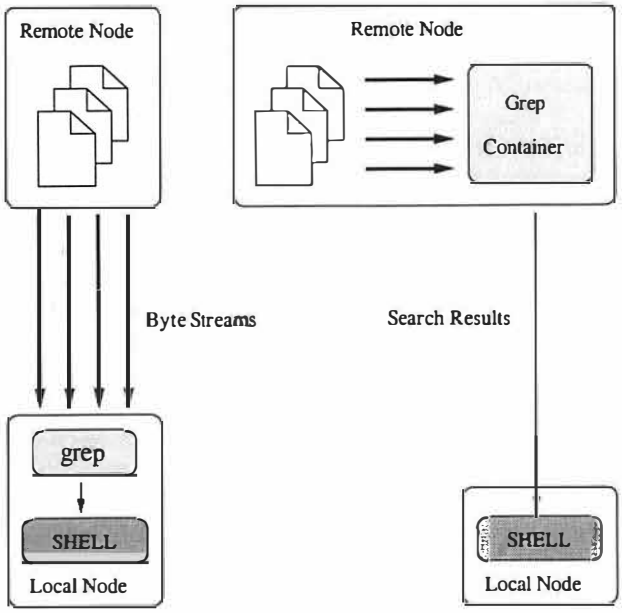


Figure 2: (a) Searching locally requires all data to be transfered to the local node. (b) Remote searching transfers only the results.

carried out to provide optimal performance. Powerful desktops connected with high-speed networks should not burden the server with these operations; they should use their own resources to complete the task. Weak devices should instead use the server computing power. Operations on data can be seen as containers that wrap a primitive data format and re-export it either as a different format or as the result of a transformation on the source.

As another example, Fig. 1 shows (on the right) a container translating MPEG to bitmaps for streaming video to a Palm Pilot device. The application can simply retrieve objects from an MPEG container and direct them to a display device, unaware of the complexity of establishing proxies and translating between data formats.

3 Architectural Design

The data service consists of two layers; a low-level system layer and a high-level user layer, as illustrated in Fig. 3. The lower layer has access to CORBA object references and includes a component to organize the storage naming hierarchy. The upper layer provides a simple user interface. We now describe both layers in the following sections.

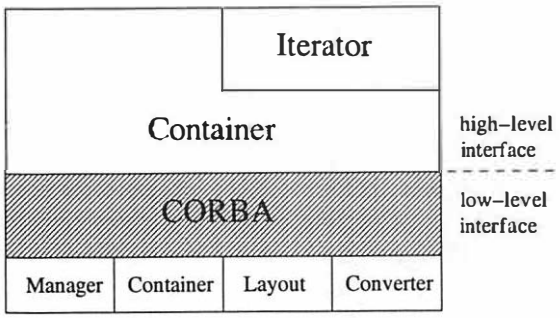


Figure 3: Layered structure of *DOS*.

3.1 User Interface Layer

The top layer consists of user level containers and iterators that hide the CORBA mechanics and reside in the local address space. Through a combination of wrapper classes and C++ templates, the user is presented with a clean and easy to use interface. Templates are used with generic programming concepts to provide a distributed generic programming model.

3.1.1 Containers

User level containers inherit from a template class that maintains a reference to the underlying CORBA container and provides methods for creation and adaptation. Container subclasses provide specialized operation for a particular container type (e.g., setting the dimensions of a slide presentation) and hide the existence of the template, providing a clean interface for application developers. If no special methods are necessary for a specific container type, the subclass is simply a wrapper and merely specifies the template parameter list. As shown in Fig. 4, the parameter list consists of the container type (C), buffer type (B), and object type (O), which are the CORBA container, transport buffer (a sequence of objects), and the indexed component object types, respectively. In addition, the subclass specifies the type of iterator to use. In this way, the application developer never sees the existence of a template, merely a particular container type (examples are given in section 3.2). The template glues

together the correct combination of components for the container to work correctly. Containers generally do not need to add specialized code, so creating a container wrapper (only specifying the parameter types) can be done in one line of code.

Templates are used to provide compile-time polymorphism of CORBA container types, thereby applying generic programming techniques to distributed objects. Different CORBA containers provide methods to get and put objects of a particular type. However, the name of the methods must adhere to a convention (*getObjects()/putObjects()*) for each container. The particular object types to be transfered are specified in the template parameter list. Therefore, the template container transfers data of a certain type when communicating with the remote ORB.

In effect, the user level container provides a consistent view of a CORBA container, although objects of different types are specified in the IDL container descriptions and are marshaled over the network. The need to use the CORBA type *Any* to transfer objects is removed and eliminates the need to typecast objects to a specific type.

3.1.2 Iterators

Iterators provide a simple interface for users to traverse the structure of the data inside a container. They maintain the current position and cache information about their respective containers. Caching specific information about the container locally reduces the need to access the remote object as often, therefore, reducing network access and latency.

Different containers require different access methods and are associated with a specific iterator type. Since containers create iterator instances, the user is forced to use the correct iterator. The syntax for obtaining an iterator is identical to the STL and examples of its use are given below.

There are two types of iterators: *ObjectIterator*s and *StreamIterator*s. *ObjectIterator*s treat the contents of a container as objects, in contrast to *StreamIterators*, that view the contents as a stream of octets. The latter are required to provide the traditional view of files, as streams of bytes, efficiently. The implementations of iterator types differ in how they detect when the iterator is at the end of a container. Subclasses provide specific methods for traversal. For example, *RandomObjectIterator* allows random placement of the iterator in the container.

Iterators are useful for retrieving remote objects incrementally [HV99] and containers hide caching of object groups. Although the container is a collection of items, the items need not all be loaded into local memory at the same time, as shown in Fig. 5. For example, when a user iterates over a collection of objects, they do not have to be individually pulled over from a remote server. Some number of objects may be prefetched and cached. The local template container plays the role of a buffer cache in standard file systems [MJ86]. If an object is requested, but is already available, it can be retrieved out of the cache. However, if the object is not available, the next group of objects may be retrieved to local memory and the current object passed to the user. The iterator hides this caching mechanism from the user; objects are handled as if they were all local.

The above described caching mechanism is used if data content is parsed remotely or on a proxy. However, if a container is resident locally, all data is transported to the local node and parsed there. Therefore, nodes with enough resources can cache the entire contents of a data source.

3.2 Interface Usage

The following examples illustrate the user interface. The first example opens a container as a stream of bytes in read-only mode. The container is then adapted to look like a container of text line objects. An iterator is then created and each line is printed to the console. Exception handling is removed for clarity.

```
ByteContainer b("MyFile", FS::Read);
LineContainer l = b.as("LineContainer");
LineContainer::iterator i;
for (i = l.begin(); i != l.end(); i++)
  cout << *i << endl;
```

It may be noted that although the containers are actually different template types, assignment is handled correctly. The *as()* method instantiates a CORBA *LineContainer* adaptor container on the local node (by default). However, the user or system may specify that it be instantiated remotely.

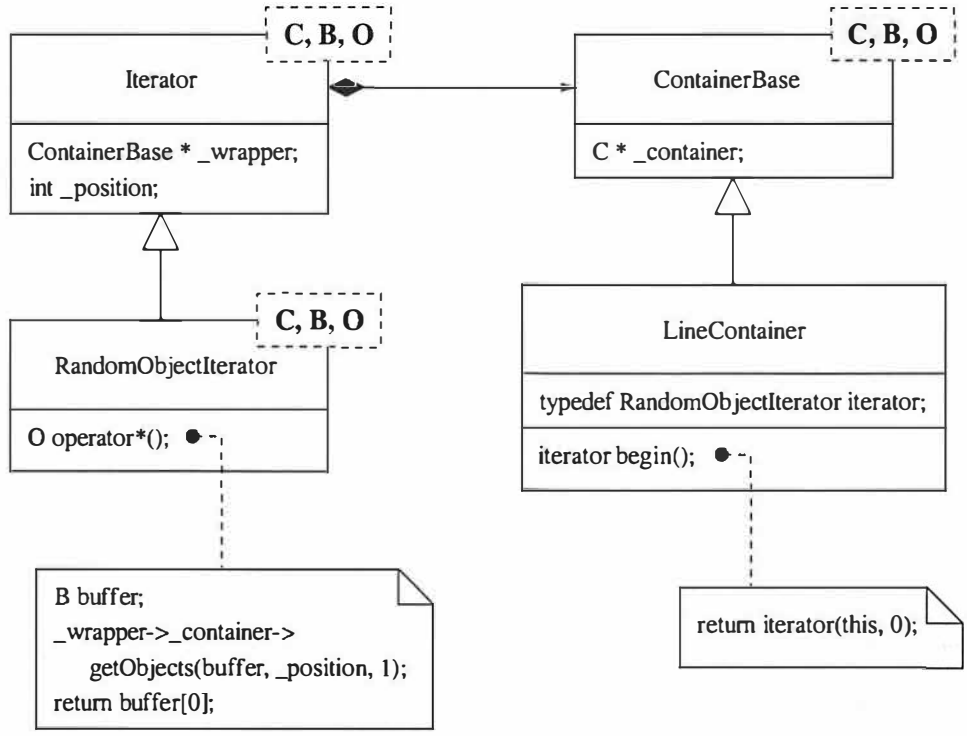


Figure 4: UML class diagram of relationship between *LineContainer* and *RandomObjectIterator* objects. Containers may retrieve groups of data objects and cache them (not shown) to reduce the number of network requests. **C** is *Container Type*; **B** is *Buffer Type*; **O** is *Object Type*.

Typically, the system uses the *as()* method to provide implicit adaptor instantiations; a container may be opened as a particular type directly, rather than having to first open it as a native container type and then specifying the adaptor. Therefore, applications can open containers in the format that they require and any adaptation/conversion is done automatically. This method is shown in the remaining examples.

The next example illustrates how a weak device may view a video sequence over a very slow network connections. Due to the limited resources available on such a device, it may be incapable of decoding and displaying MPEG video. However, the sequence may be transformed to bitmap images using a converter container and then pulled by the handheld device [HRCM00]. The container may need to handle the real-time nature of particular data sources, for example, by dropping frames if the client cannot keep up with the data source. It is the responsibility of the service to install the correct converter in the proper location, transparent to the application programmer.

```
BitmapViewer viewer;
BitmapContainer b("MyMPEG");
BitmapContainer::iterator i;
for (i = b.begin(); i != b.end(); i++)
  viewer.display(*i);
```

It may be desirable to "display" data in a format different from the source format when it is more convenient for the user. For instance, when using a computer with a small screen (e.g., a cellphone), retrieved messages may be more easily heard than read. A converter could be instantiated to present the data in the desired format.

```
AudioDevice device;
AudioContainer a("MyMailbox");
AudioContainer::iterator i = a.begin();
... get user input for message number ...
i += num;
device << *i;
```

The next example illustrates how a Palm Pilot can view a Microsoft *PowerPoint* presentation. The

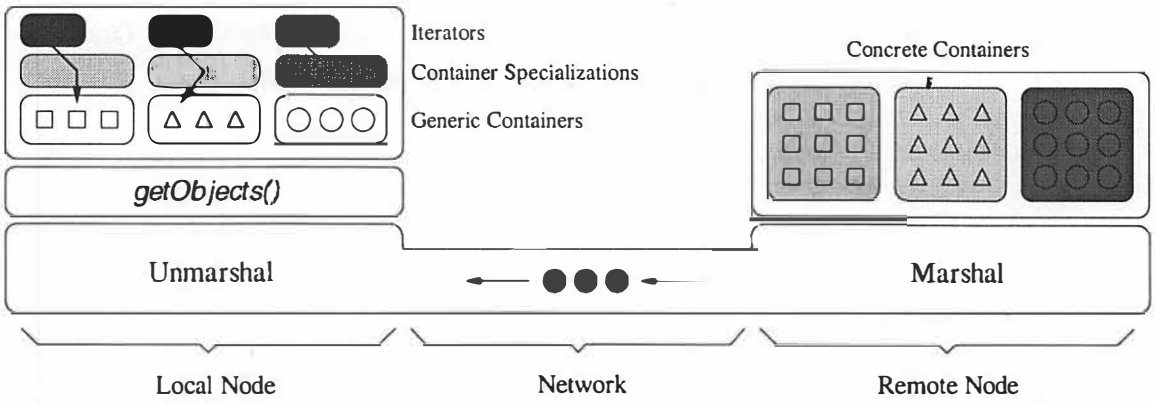


Figure 5: User-level containers are described using generic programming concepts to maximize core reuse. Typed objects are marshaled over the network. Groups of data objects can be cached in the local (generic) container.

system opens the presentation file with a *PowerPointContainer* (using OLE), which contains data objects (slides) in GIF format. It then converts the GIF slides to bitmap images using a *GIF2BitmapContainer*. The interface that the application manipulates is a *BitmapContainer*, which the *GIF2BitmapContainer* implements.

```
BitmapViewer viewer;
BitmapContainer p("MyPresentation.ppt");
BitmapContainer::iterator i;
for (i = p.begin(); i != p.end(); i++)
  ... get user input for next slide ...
  viewer.display(*i);
```

The previous examples implement different iterator types, but are used in a similar manner. The complexity of specific container and iterator creation are transparent to the user. Also notice that the containers create iterators to handle the specific data object types it holds.

3.3 System Layer

The system layer provides access to servers and servants via CORBA object references. A local component caches object references and provides name resolution support. Several types of system containers exist, which are hidden by the user level containers discussed above. The following sections describe the different types of containers available. In addition, the mechanisms that exist for locating data and creating components are discussed.

3.3.1 Containers

Containers are the main abstraction for representing data and provide methods for creation and deletion of the data objects they hold. Concrete containers are implemented using the *Gaia* component model. Each container is built as a dynamic link library (on Windows) or a shared object (on Solaris). The component model allows the service to load, create, and activate container components. Decoupling the containers from the service allows new container types to be added to the running system without interrupting current applications. There are several different container types that perform different roles in the system.

File Containers *File containers* enable access to native operating system files and directories. File containers parse data of different file types into indexed components (e.g., *DirectoryContainer*, *MailContainer*, etc). Parsing meta-data can be cached persistently for future container accesses, therefore eliminating the need to determine object boundaries each time a container is opened. This is particularly useful for containers that do not change frequently. Altering the contents of a container invalidates the cache.

There are several strategies that may be used when implementing a file container. Access mode and file size affect which strategy is employed. For example, if a container is accessed as read-only, the bytes on disk will not change. If parsing meta-data is available, access to indexed components only requires

the server to seek to a component boundary (which is included in the parse meta-data), reading in the appropriate amount of data, and sending it to the client.[1] However, if the container is accessed read-write, the byte layout on disk will probably change. Information regarding the insertion and deletion of objects in the container are cached in memory and then committed once the container is released. Alternately, the entire container can be loaded into memory (i.e., as an STL container) and insertion and deletion of objects are performed in memory. When releasing the container, the entire contents of the in-memory container is written to disk. This strategy is implemented more easily, but requires more memory. An area of future work is determining when to use a particular strategy depending on access mode and file size. For example, large files should probably not be completely loaded into memory.

Some files do not contain any well-defined structure. Such files may be represented as a stream of bytes (*ByteContainer*), thereby supporting traditional file semantics. Finally, *ByteContainer*s can be used by applications that want to bypass the type system of *DOS*, be it for backward compatibility or due to the lack of an appropriate container type.

Processor Containers Containers can represent things other than standard files and directories. *Processor containers* act as "files" with dynamic content; the "file" is created on-the-fly. For example, a *GrepContainer* may provide the ability to perform remote *grep* processing on files in a directory. This allows the computation of pattern matching to be performed at a remote location and the results transfered to the client. This not only reduces computational overhead on a weak client, but network traffic as well.

Such remote processing may be performed on the server or at a proxy node. Too much processing on a server may slow down data access by other clients [SHG98]. If performed at a proxy, the server managing the native files acts as a traditional file server (just serving byte streams). Resource consumption is therefore split between two machines; the proxy server is used for memory and CPU, while the file server is used for disk access. This is managed quite easily through CORBA, since placing a component on a particular node is only a matter of directing a particular node to instantiate the object.

[1]More than one component may be sent in one request.

Converter Containers Weak devices may not be able to render data in its original format or process containers may require data to be in a particular formats [Wir]. Conversion of content is performed via a *converter container*, which is used to transcode data to a new format. Converter containers may be created on demand or automatically, when it is determined that the original data format is inappropriate, to provide on-the-fly transcodings.

Complex conversion may require the support of several converter containers; therefore, converters can be linked together. Converters can be created on different hosts, such as the local machine, the machine maintaining the native data, or any other machine. Creating an converter inserts a component into the flow of data and changes the container interface, similar to a module in a stream [Rit84].

For example, if a converter exists that transforms Microsoft *Word* documents into ASCII text format, a *grep* could be performed on *Word* files. Since *grep* requires ASCII text as input, the *Word* file would be opened as an *ASCIIContainer*, which the system would transparently convert in order to present the file in the format that *grep* expects.

3.3.2 System Core

The system core consists of a component that maintains a cache of references to machines exporting storage and provides name resolution facilities. The core includes a prefix table mechanism[2] [WO86] and, when needed, attempts to make connections to available remote data servers or proxy servers listed in the prefix table. Each remote data server manages the data content on their respective machines and is responsible for creating CORBA container objects on that host. A server may be started on the local node (if resources are available) so that the local disk may be accessed (if available) and containers can be created locally.[3] The interface to access local and remote objects is identical, so contacting any server is merely a matter of getting an

[2]Names are paths. Path prefixes, or "mount points", are translated to object references.

[3]Mobile handheld devices would probably not launch the local server and would rely on remote servers to instantiate all containers.

object reference to the correct server. This management component is a C++ class rather than a CORBA object, since it does not need to be accessed remotely.

The local view of the storage layout (namespace) is constructed through the use of the prefix tables. The prefix tables are used for name resolution and to locate storage. When a new file, directory, or device is accessed, the local container name in the hierarchy is translated to the native name and the manager finds the correct server hosting the content. Requests are then directed towards manager components (see section 3.3.4), which are responsible for the creation/destruction of various types of containers.

3.3.3 Layout Manager

The Layout Manager stores the prefix tables that allows machines and devices to export all or a portion of their storage. This manager is implemented as a service and may provide private local storage for a group or a physical space. For example, there may be a manager running in each space. When a user with a device enters a space, the device may obtain the storage descriptions of the space to build the local storage namespace. Another, more interesting, possibility is that the mobile device exports some of its storage. Consider a room that contains a projector and presentation software. The mobile device of the user may contain the actual presentation. When the user enters the room, the device contacts the Layout Manager and informs it of which part of its storage it wishes to export and the room then adds this storage to its namespace. The user may then navigate with the presentation software, which resides in the room, to the directory containing the presentation of the user, residing on the mobile. In such a scenario, there is no need to manually transfer files; the space automatically detects the existence of a new storage device and incorporates it. Hence, the namespace (i.e., what storage the room is aware of) can change dynamically as new machines and devices enter and leave physical spaces.

3.3.4 Container Manager

Access to each data source is initiated via a *Container Manager*. These managers act as factories for container creation and are the main entry point to gaining access to object references. Once a manager has successfully created an association between a container and a native file, processor, or converter container, a reference to the container is returned.

Container Managers also assist in data content adaptation/conversion, as described above, by finding an appropriate converter and returning a new interface.[4] Conversion may be done automatically by the manager when a request to open a container type does not match the underlying data source type. It may also be performed after a container has already been opened. This procedure is illustrated in Fig. 6. In order to adapt a container interface, a container object reference is transfered to the manager performing the adaption via the *adaptInterface()* method (Fig. 6-a). The manager determines the type of the container and examines a graph to see if a possible converter exists between the container type that was passed and the desired target container type. If a suitable converter container is found, a concrete instance is created on that node (each container provides a static *create()* method to generate an instance of itself on behalf of the Container Manager factory), and the new converter container is given the original object reference. The converter uses this object reference as its data source and knows the format of data that the source provides. Therefore, the converter receives objects of one type and sends objects of another type. The object reference of the newly created converter is then returned to the client, which can use it to get and put objects of the new type the adaptor supports (Fig. 6-b). Hence, containers can be linked together and the data can change as if flows through the links. Since these converters can be placed on various nodes, they may act as proxies for weak clients.

3.3.5 Container Descriptions

In order for containers to be linked together to provide the proper conversion, a description of the containers must be available. We describe containers using XML. Each description specifies the name of the container component (i.e., the name of the library that must be loaded that contains the component), type of the container (file, processor, or converter), input data object type, output data object type, and an optional file type (expressed as

[4] The new interface has the same method names, but handles different data object types.

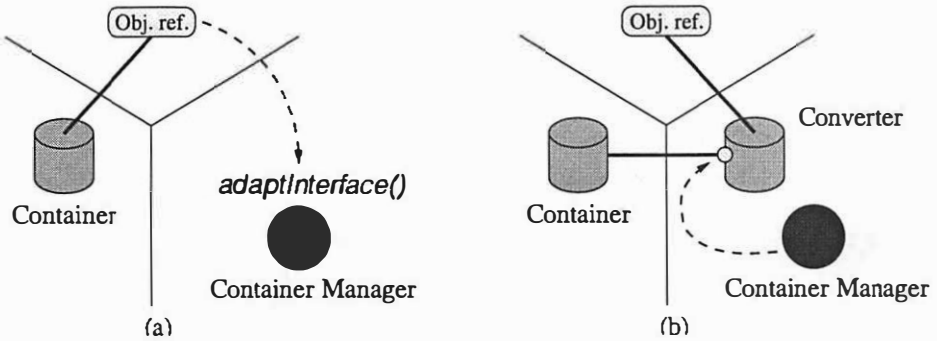


Figure 6: Container Managers enable adaptation of container interfaces.

a file extension) that the container is associated with. When a Container Manager first starts up (or when a new container type is added to the system), it reads the XML descriptions and creates a graph based on the input/output types. This graph is used to determine which containers need to be instantiated and in what order to perform a particular conversion.

4 Continuing Work

Our current implementation is based on the CORBA standard. We will port our system to the UIC composable communication core to provide a light-weight implementation that can be used on small devices, such as Palm Pilot. In addition, the UIC can be composed to provide server-side functionality. We will develop a small server-side implementation that will allow a mobile device to automatically be added to the storage namespace of a physical space once it is detected by the space. This will allow scenarios, such as the one described in section 3.3.3, to be realized.

Another issue of future work is deciding the best location to instantiate containers (i.e., where to place proxy containers). We will use the *2K* resource manager [Yam00] for load balancing and to determine if a costly operation should be performed on a proxy node. Our service contains the mechanisms to place containers on various nodes, but this decision engine must be added.

Currently, client applications must pull objects. However, there are many situations where data should be pushed out to a client. For example, if a group of users are engaged in a discussion using a whiteboard, remote users may wish to see the schematics on the board. These updates to the contents of the board should be pushed out to remote users so that they can view new drawings. We will be adding push technology to our system to facilitate such scenarios by registering callbacks with containers. Real-time data may be streamed using RTP packets. We are building mechanisms to connect containers via *streams*, that will treat RTP packets as our data objects.

5 Related Work

Our work has a resemblance to file systems in some respects. Some previous systems have treated data as groups of data, rather that contiguous bytes of of unstructured storage. Semantic file systems [GJSJ91] index data when files and directories are created and updated. They allow extraction of attributes using file-type *transducers*. Such a system provides the user with alternate views of data and a query mechanism for finding information. The *Choices* file system [Mad92] defines a framework for building different file system types. Data on secondary storage is represented as containers and is parsed and indexed depending on file type. In addition, container contents can be viewed in different ways. However, the system is not distributed and does not perform adaptation and conversions. A replacement for standard file system organization has been proposed that logically treats files as nested boxes [BA99]. Remote copy operations and converters are incorporated into the design.

The effects of mobile code were evaluated on a dis-

tributed file service [SHG98]. The cost of performing remote file operations versus increase in server load was measured. It was found that moving operations to the file server (i.e., *agrep*) is typically advantageous when client CPU power is below that of the server and network latencies are high. However, excessive computational load on the server can reduce throughput for clients simply requesting byte streams. Our service can be configured to perform such computations on a remote node when appropriate.

Our API borrows concepts from the Standard Template Library (STL) [SL94, Gla97], which defines objects for organizing collections of data. It also defines generic iterator objects, similar to the C++ stream interface [Str98], to access the data of underlying data collections. Iterators form an abstract interface to a number of different collection types. The collections are typically located in the local address space, requiring the local node to parse data into components for insertion into the collection. The *Java* stream package, (*java.io*) [Sun], defines basic streams that may be adapted to add specific functionality. However, such adaptors may only be applied locally.

Several pervasive computing projects have investigated the problem of information access and sharing in heterogeneous environments. IBM's TSpaces enhances the concept of a Tuplespace by adding consideration for heterogeneity of devices, scalability, and persistence [WMLF98]. TSpaces allow distributed applications to share information in a decoupled manner and allows a high degree of interoperability, via tuples. Their implementation includes support for access control, event notification, and efficient retrieval of information. In addition, new operators may be dynamically added to the server, which may be used immediately. This is similar to our design of allowing new container types to be spontaneously added to the system. The TSpaces project resembles a database system, where our system is more focused on adaptation of content. However, we could create a container type that was specifically tailored for tuples, which could be used as a shared data among applications.

The Infospheres project at Caltech is constructing an infrastructure for organizing task forces [Cha96]. Their goal is to build a system that allows highly dynamic groups to be rapidly assembled and share information. Other concerns are how to scale to billions of objects, restricting access to objects to authorized personnel, dealing with message delays over networks that may scale globally, and managing resources by "freezing" and "thawing" objects when needed. This research is more focused on organizing dynamic groups of people.

Jini technology enables heterogeneous devices equipped with a Java virtual machine to discover services in physical spaces [Wal]. Devices may register themselves with the Jini lookup service. Once registered, other devices may discover them and immediately use their services. Using the code mobility of Java, custom user interfaces or application may be sent to client devices to allow interaction with services or resources. Jini technology main focus is to allow devices to discover and interact with each other. Our system is more concerned with the delivery of data and adaptive content, rather than particular services.

The Document Object Model (DOM) is an object-oriented model to represent documents as a tree of nodes [CBNW]. Interfaces are available to traverse and manipulate the tree to gain access to structured data. The DOM interfaces are typically used as a result of parsing XML documents. Such documents can encompass an array of object types. We were more concerned with groups of similar objects, which simplifies our user interface. We could, however, create a container that resembles the functionality of DOM.

The work most similar to ours is that of the Ninja project from UC Berkeley [GWvB+00]. The Ninja architecture defines four main components: bases, units, active proxies, and paths. Bases are manifested as a cluster of workstations that provide scalability, fault tolerance, and concurrency. Units comprise the myriad of devices that may be connected to the infrastructure. The active proxies provide adaptation of content (similar to our containers), and are the result of previous research in data distillation using the TACC [Fox] model to perform on-the-fly data transformations [FGBA96]. Transcoding data formats was found to greatly increase the performance of certain applications [FGG+98]. The last component, paths, constructs flows of data that may be transformed while passing through different components, using their active proxies. These are similar to our channels. Our methodology is slightly different, in that we have leveraged the features of CORBA and its services and approach the problem from an operating system point of view, where the Ninja project takes a Java-centric Internet-service

approach. We have also focused on the user interface, to allow simple data access and ease application development.

6 Conclusions

DOS provides the data transfer mechanism in *Gaia*, an operating system for physical spaces. *DOS* is able to alter behavior based on knowledge of computing device characteristics and location. Data is represented by containers and access is gained using iterators. Modeling files and directories as containers unifies the interface for data distribution in our system; the interface is also used to model data operations, conversions, and proxies. Containers may be connected together as modules that can act on data passing through them. The system is aware of its environment and has the mechanisms to instantiate objects in the proper locations to optimize performance and provide load balancing.

We have hidden the details of CORBA from the developer by using C++ templates and wrapper classes and have applied generic programming concepts to distributed objects. Objects of a particular type are marshaled over the network and typecasting becomes unnecessary. User-level containers and iterators are defined as template classes that combine the proper components together to allow type-safe remote access to objects. Templates have made the construction of user-level containers trivial.

DOS is a dynamic flexible system, in contrast to typical distributed file systems, that are designed for a particular operating environment. The dynamic nature of our data service makes it well-suited for heterogeneous environments, prevalent in pervasive computing.

7 Acknowledgments

We would like to thank Fabio Kon and Manuel Román for helpful discussions regarding the design of our data service. We would also like to thank the anonymous reviewers for many valuable suggestions for improving this paper.

8 Availability

Resources for *DOS* and *Gaia* are available at:

`http://choices.cs.uiuc.edu/gaia`

References

[Abo99] G. D. Abowd. Classroom 2000: An experiment with the instrumentation of a living educational environment. *IBM Systems Journal*, 38(4), 1999.

[BA99] Francisco J. Ballesteros and Sergio Arevalo. The Box: A replacement for files. In *IEEE Hot Topics on Operating Systems (HotOS-VII)*, Rio Rico, AZ (USA), 1999.

[CBNW] Mike Champion, Steve Byrne, Gavin Nicol, and Lauren Wood. Document Object Model (Core) Level 1. `http://www.w3.org/TR/REC-DOM-Level-1/level-one-core.html`.

[Cha96] K. Mani Chandy. Caltech Infosperes Project Overview: Information Infrastructures for Task Forces. `http://www.infospheres.caltech.edu`, November 1996.

[FGBA96] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to Network and Client Variation via On-Demand Dynamic Distillation. In *ASPLOS-VII*, Boston, MA, October 1996.

[FGG+98] Armando Fox, Ian Goldberg, Steven D. Gribble, David C. Lee, Anthony Polito, and Eric A. Brewer. Experience With Top Gun Wingman: A Proxy-Based Graphical Web Browser for the 3Com PalmPilot. In *IFIP International Conference on Distributed Systems Platformas and Open Distributed Processing*, Lake District, UK, September 1998.

[Fox] Armando Fox. The Case for TACC: Scalable Servers for Transformation, Aggregation, Caching, and

Customization. Qualifying Exam Proposal.

[Gai00] Gaia Research Team. Gaia: Enabling Active Spaces. `http://choices.cs.uiuc.edu/gaia`, 2000.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Longman, Inc., 1995.

[GJSJ91] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole Jr. Semantic File Systems. In *Proceedings of the 13th Symposium on Operating System Principles*, pages 16–25, 1991.

[Gla97] Graham Glass. The Java Generic Libarary. *C++ Report*, 9(1):70–74, January 1997.

[GWvB+00] Steven D. Gribble, Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Josheph, R. H. Katz, Z. M. Mao, S. Ross, and B. Zhao. The Ninja Architecture for Robust Internet-Scale Systems and Services. *Special Issue of Computer Networks on Pervasive Computing*, 2000. (to appear).

[Hew] Hewlett Packard Company. Cooltown. `http://www.cooltown.hp.com`.

[How88] John H. Howard. An Overview of the Andrew File System. In *USENIX Winter Conference*, pages 23–26, Dallas, Texas, February 9-12 1988.

[HRCM00] Christopher K. Hess, David Raila, Roy H. Campbell, and Dennis Mickunas. Design and Performance of MPEG Streaming to Palmtop Computers. In *Multimedia Computing and Networking 2000 (MMCN00)*, San Jose, CA, January 25-27 2000. ACM.

[HV99] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++.* Addison Wesley Longman, Inc., 1999.

[KRL+00] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, New York, April 2000.

[Mad92] Peter William Madany. *An Object-Oriented Framework for File Systems.* PhD thesis, University of Illinois at Urbana-Champaign, June 1992.

[Mic] Microsoft Corp. Easyliving. `http://www.research.microsoft.com/easyliving`.

[MIT] MIT Media Lab. Smart Rooms. `http://ali.www.media.mit.edu/vismod/demos/smartroom`.

[MJ86] Bach Maurice J. *The Design of the UNIX Operating System.* Prentice-Hall Software Series. Prentice Hall, 1986.

[MS96] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Addison Wesley Longman, Inc., 1996.

[Mus89] David R. Musser. Generic Programming. *Lecture Notes in Computer Science 358*, pages 13–25, 1989. Springer-Verlag.

[RC00] Manuel Roman and Roy H. Campbell. GAIA: Enabling Active Spaces. In *9th ACM SIGOPS European Workshop*, Kolding, Denmark, September 17-20 2000.

[Rit84] Dennis M. Ritchie. A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal*, 63(2):1897–1910, October 1984.

[SC99] Douglas C. Schmidt and Chris Cleeland. Applying Patterns to Develop Extensible ORB Middleware. *IEEE Communications Magazine*, 1999. available at `http://www.cs.wustl.edu/~schmidt/ACE-papers.html`.

[SGK+85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130, Portland, Oregon, June 1985.

[Sha86] Marc Shapiro. Structure and Encapsulation in Distributed Systems: the Proxy Principle. In *International Conference on Distributed Computing Systems (ICDCS'86)*, Cambridge, MA, May 19-23 1986.

[SHG98] Tammo Spalink, John H. Hartman, and Garth Gibson. The Effect of Mobile Code on File Service. Technical Report TR98-12, Department of Computer Science, Universiy of Arizona, November 1998.

[SL94] Alexander Stepanov and Meng Lee. The Standard Template Library. Technical report, HPL-94-34, April 1994. revised July 7, 1995.

[Str98] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesely Longman, Inc., 3rd edition edition, 1998.

[Sun] Sun Microsystems, Inc. Java 2 Platform API Specification. `http://www.java.sun.com/products/jdk/1.2/docs/api`.

[The98] The Object Management Group (OMG). The Common Object Request Broker: Architecture and Specification. `http://www.omg.org/library/c2index.html`, February 1998.

[UBI00] UBICore, LLC. UBICore web page. `http://www.ubi-core.com`, 2000.

[Wal] Jim Waldo. Jini Architecture Overview. `http://java.sun.com/products/jini/whitepapers`.

[Wei93] Mark Weiser. Some Computer Science Issues in Ubiquitous Computing. *CACM*, 36(7):74–84, 1993.

[Wel92] Brent Welch. A Comparison of the Sprite and Vnode File System Architectures. In *USENIX File System Workshop Proceedings*, pages 29–44, Ann Arbor, MI, May 21-22 1992.

[Wir] Wireless Application Protocol Forum, Ltd. Wireless Application Protocol Architecture Specification. `http://www.wapforum.org`.

[WMLF98] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T Spaces. *IBM Systems Journal*, August 1998.

[WO86] Brent B. Welsh and John K. Ousterhout. Prefix Tables: A Simple Mechanisms for Locating Files in a Distributed System. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, Cambridge, MA, 1986. IEEE Computer Society Press.

[Yam00] Tomonori Yamane. The Design and Implementation of the 2K Resource Management Service. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, February 2000.

HBench:JGC – An Application-Specific Benchmark Suite for Evaluating JVM Garbage Collector Performance

Xiaolan Zhang and Margo Seltzer
Division of Engineering and Applied Sciences
Harvard University

Abstract

As Java becomes a viable platform for server applications, performance becomes a greater concern. An important aspect of Java Virtual Machine performance is its dynamic memory management system (*garbage collection* or *GC*). Traditional GC benchmarking often focuses on a set of fixed applications. As a result, when an actual application's memory behavior differs from that of the standard benchmarks, the benchmark results do not help the user judge which GC implementation suits her application the best. In this paper, we present HBench:JGC, an application-specific benchmarking suite, based on the idea that a system's performance be measured in the context of a specific application. HBench:JGC employs a methodology that characterizes the application memory usage and the GC implementation independently and carefully combines both characterizations to form a single metric that reflects a particular application's performance in the presence of a particular GC implementation. We evaluate our approach on Sun Microsystems's JDK1.2.2 classic JVM with a sequential mark-sweep GC. Our results demonstrate HBench:JGC's unique predictive power and its ability to provide meaningful metrics that lead to a better understanding of GC performance.

1. Introduction

In recent years, there has been a rapid increase in the adoption of Java technology in a variety of environments, ranging from JVMs embedded in web-browsers to high-performance server products. As Java becomes a viable platform for server applications, performance becomes a greater concern. An important piece of Java Virtual Machine performance is its dynamic memory management system (*garbage collection* or *GC*). Historic data show that it is quite common for garbage collection to account for 20% or more of an application's total running time [9]. Sometimes garbage collection is the performance bottleneck. Understanding GC performance and selecting the right GC implementation, therefore, can lead to significant savings in the total running time of the application.

The traditional GC benchmarking approach is to pick a set of programs, run them with different GC algorithms, and compare the total elapsed times. This approach has been used by Smith and Dorisett, as well as by Zorn [12][15]. This approach is inadequate, since the optimal GC algorithm varies with the application [15], and the set of benchmark programs may not represent the actual memory behavior of the application of interest.

Another approach to benchmarking and selecting GC algorithms for a given application is to manually construct a small program that models the memory behavior of the application in question and when run, produces the same memory footprint. This approach requires a high level of skill and is error-prone, especially when the application's memory behavior is complicated.

HBench:JGC is a benchmark suite that allows one to measure GC performance in the context of the applications in which users are interested without having to model the applications manually. The underlying principle is to separate the characterization of application memory usage from that of the GC implementation. HBench:JGC includes a GC-independent profiler that traces an application's memory behavior. It uses a set of microbenchmarks to measure the performance of a GC implementation in an application-independent way. The two characterizations are then fed to an analyzer, which calculates the predicted GC time. Figure 1 depicts the schema of HBench:JGC. HBench:JGC has the added advantage that one can use it to predict an application's garbage collector performance on a target GC implementation without actually running the application with the particular collector, as long as its performance characteristics are available. The GC-independence of the application characterization facilitates this unique flexibility. We

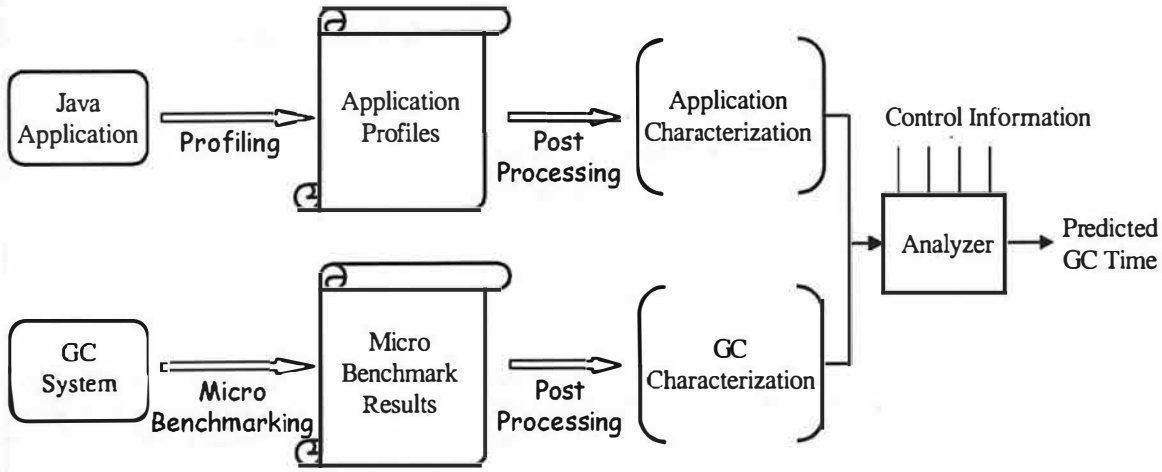


Figure 1. Schematic View of HBench:JGC Process

can predict the application's performance on different GC implementations by feeding performance characteristics of different GC implementations to the analyzer.

Section 2 describes the design of HBench:JGC in detail. Section 3 describes its prototype implementation. Section 4 presents experimental results on applying HBench:JGC to evaluating GC performance. Section 5 discusses open issues and future work. Section 6 describes related work and Section 7 concludes.

2. HBench:JGC Design

HBench:JGC is part of HBench, an application-specific benchmarking framework designed to address the problem that standard benchmark results do not reflect a particular application's performance on a particular system [11]. HBench:JGC is based on HBench's vector-based methodology. The principle behind the vector-based methodology is that a system's performance is determined by the performance of the individual primitive operations that it supports and that an application's performance is determined by how much it utilizes the primitive operations of the underlying system. The running time of a given application can be estimated by carefully combining the two characterizations. A simple form of this combination process would be to add up the costs of all primitive operations executed by the application. By separating characterizations of the application from that of the underlying system and by incorporating application characteristics into the benchmarking process, HBench can provide performance metrics that reflect the expected behavior of a particular application on a particular platform, as well as allow meaningful comparisons between different platforms.

Although originally designed as part of the HBench:Java benchmark suite [14], the methodology of HBench:JGC described in this paper is applicable to GC implementations for other languages such as Lisp, Scheme, Smalltalk, and C++.

2.1. GC Characterization

2.1.1. Basic GC Concepts

Like all memory management systems, a garbage collector implementation supports two primitive operations, namely, object allocation and reclamation.

The garbage collector manages the collection of free space from which new objects are allocated. The free space can be represented as a list of free blocks, a single chunk of contiguous space, or a combination of the two.

When the allocator fails to satisfy an allocation request, it initiates a garbage collection run. A garbage collection run typically starts with a *marking* phase, when live objects are identified and marked. This phase may be followed by one or more phases (typically called the *sweep* phases) that free the space occupied by the dead objects, making it available for allocation. *A non-copying* collector does not move the live objects, whereas a *copying* collector typically compacts the live objects to one end of the heap in order to create a large contiguous free space at the other end of the heap. Examples of non-copying col-

lectors include the most widely adopted mark-sweep garbage collector [1] and its variants. Examples of copying collectors include the Lisp 2 collector [8], which is a mark-compact collector, and Cheney's two-space copying collector [3]. For a complete treatment of this topic, readers are encouraged to refer to the book by Jones et al. [7].

2.1.2. A GC Implementation Taxonomy

Independent of the GC algorithms (e.g., copying vs. non-copying), we can classify GC implementations according to the four attributes described in Table 1. The first attribute represents the axis between stopping all execution for garbage collection and running the collector completely in parallel with program execution [2]. The second attribute describes the internal architecture of the collector itself, whether it is sequential (single-threaded) or parallel (multi-threaded). The third attribute describes the granularity of collection, whether collection occurs in a single, complete pass (batch-oriented) or whether just some of the available memory is reclaimed during each iteration (incremental). The fourth and last attribute distinguishes generational garbage collectors [10] from non-generational collectors. Generational collectors implement a set of heaps that are cleaned with varying frequency depending on the age of the objects stored in the heap. Each heap corresponds to a different age group.

Attributes of GC Implementations
Stop-the-world ↔ Concurrent
Sequential ↔ Parallel
Batch ↔ Incremental
Non-generational ↔ Generational

Table 1. GC Implementation Techniques

The four attributes in the taxonomy are largely orthogonal, with a few exceptions. For example, a GC algorithm can be both stop-the-world and parallel, but it cannot be both concurrent and batch mode.

In this paper we consider only sequential, stop-the-world, batch-mode and non-generational garbage collectors. We chose to start with this type of collector because it involves the fewest variables and thus allows faster prototyping of the analytical models and more controllable experimentation. Furthermore, this type of collector is still in wide use. For example, Sun's standard JDK1.1 and JDK1.2 Java Virtual Machines use this type of collector. Section 5 discusses how we envision enhancing our approach to cope with concurrent, parallel, incremental and generational garbage collectors.

2.1.3. Object Allocation

For a given memory management algorithm, the cost of object allocation is typically determined by the following two factors:

1. the size of the allocation,
2. the state of the heap, such as the number of free blocks and their sizes.

We can represent this cost with a function C_{alloc}(*heap_state, allocation_size*). Depending on the memory management algorithm, C_{alloc} carries different forms. In the case of copying garbage collectors, the free space is a contiguous area, and allocation can be implemented by a simple pointer advancement. Therefore, in the case of a copying collector, C_{alloc} is a constant function. In the case of non-copying collectors, such as a non-copying mark and sweep collector, the allocation time depends on the state of the free-block lists maintained by the collector. If we characterize the heap state with simple statistical measures, such as a normal distribution with a given mean and standard deviation, or a uniform distribution with a given range, we can represent C_{alloc} in a concise way. Furthermore, we can measure C_{alloc} using microbenchmarks that initialize the heap according to the statistical measures.

2.1.4 Object Reclamation

An interesting aspect of garbage collection performance is that the cost of dead object reclamation depends on the amount of live data on the heap, since the way a garbage collector identifies live objects is to traverse the connected object graph from a set of root objects.

We divide the cost of object reclamation into three parts: the fixed cost (C_{fixed}), the per-live-object cost (C_{live}), and the per-dead-object cost (C_{dead}). C_{fixed} corresponds to the fixed cost associated with a garbage collection run, such as the initialization of data structures. C_{fixed} normally depends only on the heap size. C_{live} is the overhead measured per live object (objects that survive

the collection). For non-copying collectors, C_{live} is typically constant. For copying collectors, C_{live} is a function of the size of live objects, as live objects are compacted (copied) at the end of a collection run. C_{dead} corresponds to the per-object cost of releasing the space of a dead object. In most cases, this involves updating bookkeeping information for the freed object, and thus C_{dead} is usually constant for a given collector algorithm. In summary, the cost of object reclamation can be represented by three functions, $C_{fixed}(heap\_size)$, $C_{live}(object\_size)$, and C_{dead}. Let N_l be the distribution function of the sizes of live objects, i.e. $N_l(s)$ is the number of surviving objects with size s. Let N_d be the distribution function of dead object sizes. The total cost of garbage collecting a heap of size h can then be calculated using the following formula (1):

$$T_{GC} = C_{fixed}(h) + \sum_{s} C_{live}(s) \cdot N_l(s) + C_{dead} \sum_{s} N_d(s)$$

The above reasoning makes the simplifying assumption that every live object is traversed exactly once during marking. For cases where an object is referenced by several live objects, the object will be visited multiple times by the collector. We characterize this additional cost by adding a second variable, d_i, the *fan-in degree* of an object, in the per-live-object overhead function C_{live}. The middle term of the formula thus becomes:

$$\sum_{s} \sum_{d_i} C_{live}(s, d_i) \cdot N_l(s, d_i)$$

The situation is further complicated by the fact that certain copying collectors need to update an object's references, if the objects it points to are copied to a different place. We characterize this additional cost by adding yet another variable, d_o, the *fan-out degree* of an object, in the per-live-object overhead function C_{live}. The middle term now becomes:

$$\sum_{s} \sum_{d_i} \sum_{d_o} C_{live}(s, d_i, d_o) \cdot N_l(s, d_i, d_o)$$

The difficulty of characterizing object reclamation costs lies in deriving the three cost functions C_{fixed}, C_{live}, and C_{dead} using results from microbenchmarks. Our experience indicates that the simplified formula (1) for estimating GC time works well in practice for a mark-sweep GC algorithm. In the future, we will include the refinements discussed above if necessary.

2.2. Application Characterization

The following metrics describe an application's memory usage behavior:

1. Object allocation rate (both in terms of the number of objects and the number of bytes);
2. Object death rate (both in terms of the number of objects and the number of bytes);
3. Object age (the time an object remains alive);
4. Connectivity of the live object graph, i.e., the number of references to an object (*fan-in degree*) and the number of references it contains (*fan-out degree*).

Some of the metrics, such as object allocation rate, can be obtained quite easily. Some other metrics, such as object age, are difficult to measure and can only be estimated using profiling tools.

One significant challenge in characterizing an application's memory behavior is that of GC (and JVM) independence. For example, if we use the number of objects per second as the unit for object allocation speed, it is not portable to other JVM or GC implementations, as this unit is system dependent. To solve this problem, we use objects per bytecode as our basic unit for both object allocation rate and object death rate.

2.3. Predicting GC Time

Object allocation cost is an important part of the performance metric of GC systems. It is, however, not directly measurable for a given application. As a first step, this paper focuses on predicting the time the application spends on garbage collection, or the time between the start and finish of a garbage collection run. Unless otherwise specified, GC time refers to the cost of object reclamation, and does not include allocation costs.

The total GC time of an application can be determined by two factors: the number of GC runs and the time for each GC run.

With the knowledge of object allocation rate and object death rate, one can estimate the amount of live data at a given execution point, from which one can then calculate the number of GCs deterministically, assuming a heap that is fixed-size or one whose growth policy is known a priori.

The time for each GC run can be estimated using formula (1) described in section 2.1.3. The total GC time is the sum of times of all individual GC runs.

3. HBench:JGC Implementation

As depicted in Figure 1, the major components of HBench:JGC are: the profiler that traces an application's memory behavior, the set of microbenchmarks whose measurement results form the characterization of the given garbage collection implementation, and finally, the analyzer that estimates the GC time given both application and GC characterizations. The following three subsections describe each component in more detail.

3.1 Profiler

Sun Microsystems's JDK 1.2.2 provides an interface called the Java Virtual Machine Profiling Interface (JVMPI) [6] that allows one to attach a profiling agent to the JVM at startup time. The agent can register for events in which it is interested through callback functions and intercept the events as they occur.

We are interested in the following events: GC start and finish, object allocation, free and move, heap dump and object dump. Object allocation and free events can be used to estimate object lifetimes and the number of free/live objects at a given execution point. Heap dumps help determine the object connectivity such as fan-in and fan-out degrees. Our current implementation includes all the events except heap and object dump.

3.2. Microbenchmarks

The goal of microbenchmarking is to measure the fixed and per-object costs of memory reclamation. Our first microbenchmark deals with singular linked list data structures. We are in the process of creating microbenchmarks that model more complicated object types with different fan-in and fan-out degrees.

The microbenchmark first populates the heap with an array of linked lists of objects. The size of array, the length of the list, and the object size can all be dynamically configured with command-line options. The microbenchmark then explicitly invokes garbage collection at three different times:

1. When all objects on the heap are alive;
2. When all objects on the heap are reclaimable, i.e., after the microbenchmark sets the pointers to the heads of the linked lists to null;
3. When the heap is entirely empty, i.e., after the GC following step 2.

To measure C_{fixed}, we run the microbenchmark with different heap sizes, fixing the other two parameters. We then plot the GC times measured in step 3 above against the heap sizes. The resulting regression formula is the approximate function for C_{fixed}.

Similarly, to measure C_{live}, we run the microbenchmark with a varying numbers of objects, fixing the other two parameters. The GC times measured in step 1 above are then plotted against the number of objects for a given object size s and the resulting regression function defines $C_{live}(s)$. Since C_{live} might also depend on object sizes, we again repeat the microbenchmark for different object sizes.

The same process is performed to measure C_{dead}, except that in this case the GC times of step 2 are used.

3.3. Analyzer

Given both the application and GC characterizations, the analyzer tries to estimate the time the application spends on garbage collection. The analyzer also needs certain configuration information, such as the heap size, in order to determine the total GC time. Note that heap sizes may change dynamically. For example, if the memory system cannot satisfy the allocation request even after a GC, or if the percentage of free space is below a certain threshold, the heap is expanded. The policies as to when and how much to expand the heap should be specified to the analyzer.

CPU	Memory (MB)	Operating System	JVM Version	GC Algorithm
Pentium Pro 200MHz	128	Windows NT 4.0	1.2.2 Classic	Mostly mark-sweep
Pentium III 550 MHz	256	Windows 2000		
UltraSPARC IIi 333 MHz	128	Solaris 7		

Table 2. Test Configurations

4. Experimental Results

4.1. Experimental Setup

We ran our experiments on Sun Microsystems's JDK1.2.2 classic version on three different machine configurations. Table 2 shows the hardware properties.

Sun Microsystems' JDK1.2.2 classic JVM uses a mark-sweep (with compaction) collector. Mark-sweep collection is one of the classical garbage collection algorithms that remains in wide usage today. Due to its conservative nature, it is popular for type-unsafe languages such as C/C++. The collector of the JDK1.2.2 classic JVM is a variation of the classical mark-sweep collector — it occasionally moves live objects around the heap. Although compaction does not occur often for the applications we tested, it does generate some uncertainties that make it harder to predict the GC time.

We use Java applications included in the SPECJVM98 benchmark suite [13] to evaluate the predictive power of our approach. Most SPECJVM98 applications induce extensive GC activities, except _222_mpegaudio, which is excluded from our set of test applications. Table 3 shows the number of bytes allocated by each test application quoted from the benchmark's documentation. The actual numbers appear to differ but the magnitude is the same.

SPEC Application	Allocation (MB)
_201_compress	334
_202_jess	748
_209_db	224
_213_javac	518
_227_mtrt	355
_228_jack	481

Table 3. GC Activity of Test Applications

4.2. Microbenchmark Results

We report the GC times of the three steps described in Section 3.2. Unless otherwise specified, all data points reported in this section are means of 10 runs of the microbenchmark. In most cases, the standard deviation is within 1%.

4.2.1. GC on Empty Heap

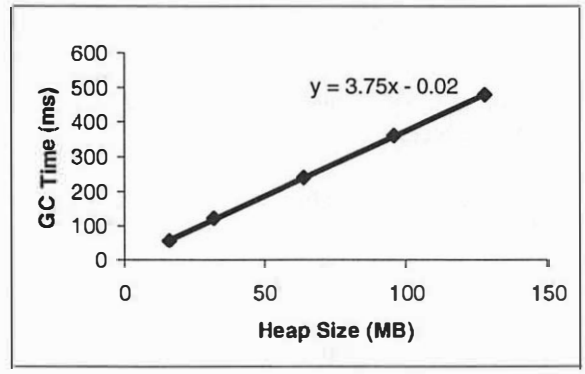


Figure 2. GC Time of Empty Heap on Sun SPARC. We use an object size of 28 bytes, and 512 lists each with 512 objects. The number of objects and the size of objects remain fixed as the total heap size varies.

Figure 2 shows the garbage collection times of an empty heap (see step 3 in section 3.2) on the Sun SPARC workstation. The regression formula indicates that GC times of empty heaps are linearly dependent on the size of the heap and that the per-megabyte cost of an empty heap GC for this particular GC implementation is 3.75ms. The y-intercept (0.02) is negligible. We therefore derive the following formula for $C_{fixed}(h)$ (described in Section 2.2) for this GC algorithm:

$$C_{fixed}(h) = 3.75 \cdot h \,,$$

where h is the size of the heap in megabytes. The value of the slope (3.75) remains the same (variations

within 5%) for different object sizes and numbers of objects.

Similar results were obtained for the other machine configurations, albeit with a different slope value.

4.2.2. GC on Fully Reclaimable Heap

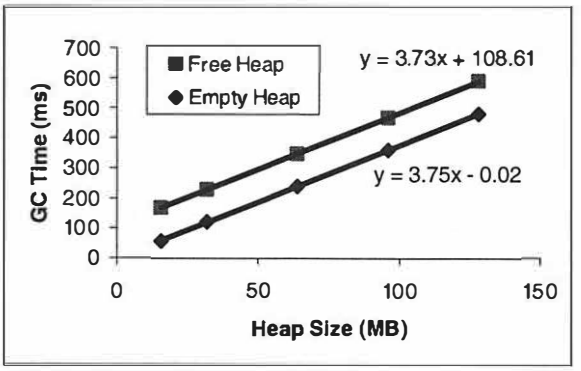
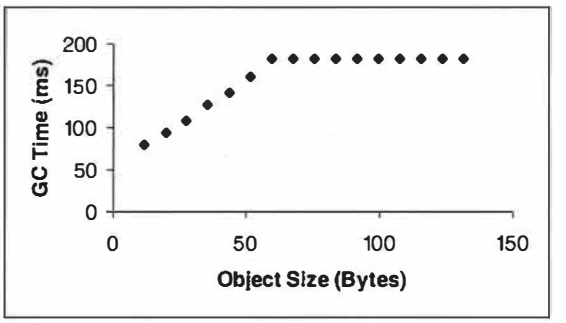


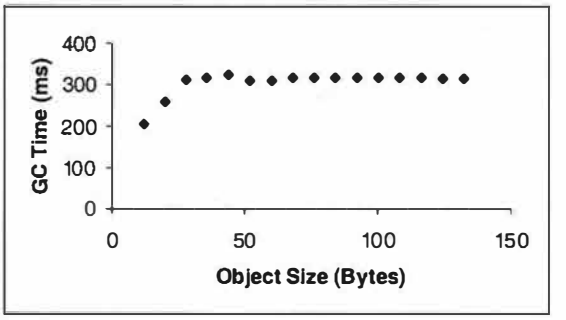
Figure 3. GC Time of Fully Reclaimable Heap With Respect to Heap Size on Sun SPARC. We use an object size of 28 bytes, and 512 lists each with 512 objects. The number of objects and the size of objects remain fixed as the total heap size varies.

Figure 3 shows the garbage collection times of a fully reclaimable heap (see step 2 in section 3.2). The GC time again shows a linear dependence on the size of the heap, and the slope value (3.73) is close to the slope value of C_{fixed} (3.75). If we remove the fixed cost C_{fixed} (empty heap), the remaining time is essentially independent of heap size. Since all objects on the heap are free and are reclaimed by the collector, this remaining time, when divided by the number of dead objects, represents the per-dead-object cost C_{dead}. In this particular case, C_{dead} takes on a value of 108.6/(512*512), or 0.4 ns/object. Again, similar results are observed from runs on the other machine configurations.

Theoretically, C_{dead} is independent of object size, since dead objects are neither scanned nor copied. However, to our surprise, our measurements suggest that C_{dead} is indeed dependent on object size. Figure 4(a) shows the results on the Sun SPARC workstation. The GC time seems to grow as the object size increases, until the object size hits 60 bytes, and stays at around 180ms thereafter. We do not have a conclusive explanation for this behavior but we hypothesize that the dependence on the object size is due to memory cache effects.



(a). Results on Sun SPARC



(b). Results on Pentium Pro

(c). Results on Pentium III

Figure 4. GC Time (Excluding Fixed Overhead) of Fully Reclaimable Heap with Respect to Object Size. The GC time is calculated from the regression formula as shown in Figure 3.

Our experiments on the other two machine configurations seem to confirm our hypothesis. Figures 4(b) and 4(c) show the results on the Pentium Pro and Pentium III machines respectively. In both cases, C_{dead} shows similar dependence patterns. C_{dead} is independent of object size, except when the object size is less than 28 bytes. The memory effects seem to be smaller for these two configurations than for the Sun SPARC workstation.

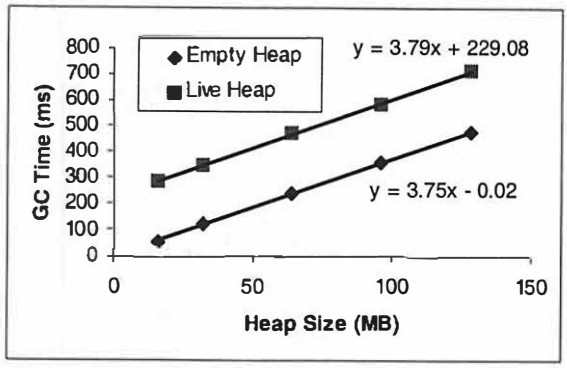
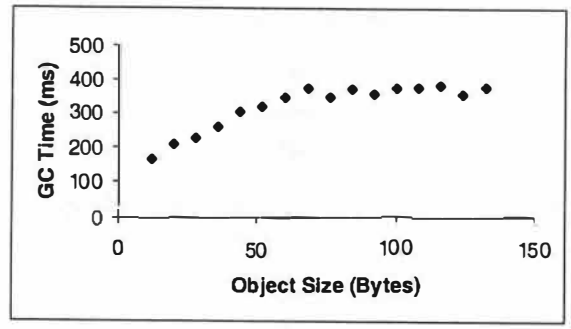


Figure 5. GC Time of Fully Live Heap with Respect to Heap Size on Sun SPARC. We use an object size of 28 bytes, and 512 lists each with 512 objects. The number of objects and the size of objects remain fixed as the total heap size varies.

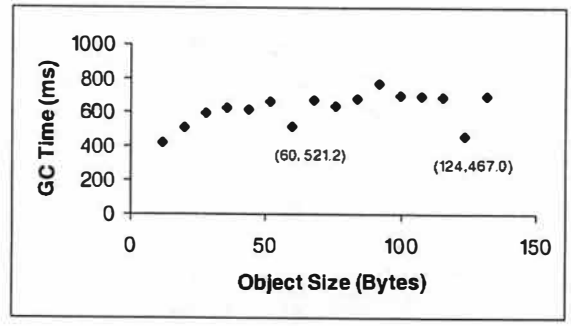
4.2.3. GC on Fully Live Heap

Figure 5 shows the garbage collection times of a fully live heap (see step 1 in section 3.2). In this case, all objects on the heap are live and survive the garbage collection. Similar to the case of a fully reclaimable heap, the GC time shows a linear dependence on the size of the heap. If we exclude the fixed cost C_{fixed}, the remaining time is independent of heap size. The GC time, when divided by the number of total objects on the heap, yields the per-live-object cost C_{live}. In this particular case, C_{live} takes on a value of 229.1/(512×512), or about 0.9ns/object. Again similar results are observed from runs on other machine configurations.

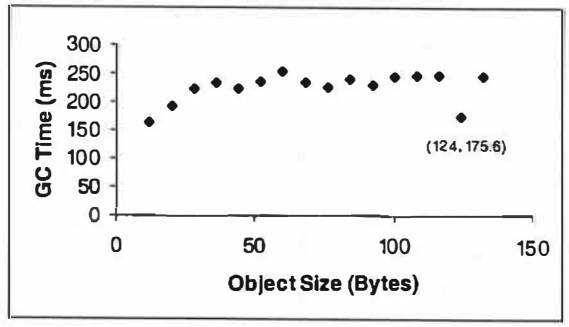
Figures 6 (a), (b) and (c) show C_{live} as a function of object size on the Sun SPARC workstation, the Pentium Pro machine, and the Pentium III machine, respectively. We observe patterns similar to those of the fully reclaimable heap case, albeit with different threshold values. For the Sun SPARC workstation case, the value of C_{live} seems to grow as the object size increases, until the object size hits 60 bytes and stays at approximately 380ms thereafter. For the Pentium Pro machine case, the value of C_{live} seems to oscillate between 600ms and 700ms after the object size hits 28 bytes. Similarly, for the Pentium III machine case, the value of C_{live} oscillates between 220ms and 250ms after the object size hits 28 bytes. Since no objects are copied, C_{live} should be independent of object size. We therefore attribute this observed dependence on object size to memory cache effects. This effect is also detected with two anomalous data points for the Pentium Pro configuration: at object sizes of 60 bytes and 124 bytes. There is also a similar anomalous data point for the Pentium III at object size of 124 bytes. We are still investigating what exact memory effect causes the anomalies.



(a). Results on Sun SPARC



(b). Results on Pentium Pro



(c). Results on Pentium III

Figure 6. GC Time (Excluding Fixed Overhead) of Fully Reclaimable Heap with Respect to Object Size. The GC time is calculated from the regression formula as shown in Figure 5.

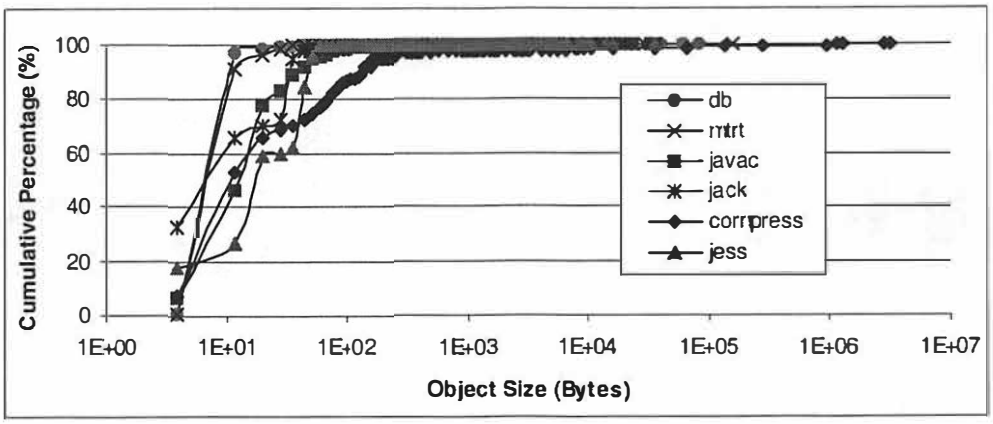


Figure 7. Cumulative Object Size Distribution in Number of Objects

4.3. Predicting GC Time

In this section we demonstrate how the microbenchmark results can be used to predict garbage collection time for a given Java application.

First, we calculate the values of the three functions that characterize a GC algorithm, namely, C_{fixed}, C_{live}, and C_{dead}. Table 4 shows the coefficient values of the three functions for the JVM on the Sun SPARC workstation. For objects with size larger than 132 bytes, the values for 132 bytes are used.

Next we obtain characterizations of the applications' memory behavior. Our current profiler implementation generates information such as the number of live objects, the number of dead objects, and the object size distribution. Assuming that live and dead objects have the same size distribution, we can approximate the GC time function T_{GC} (section 2.1.4) with the following formula

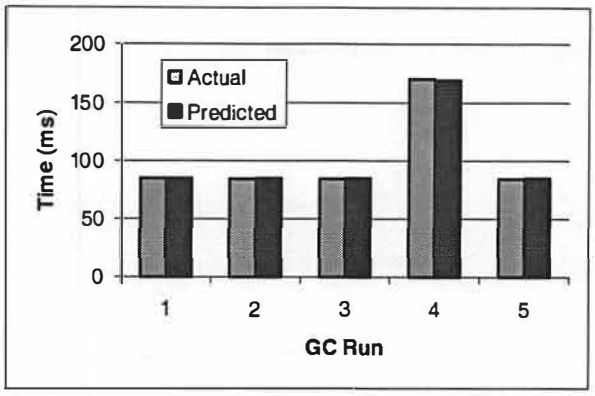
$$T_{GC} = C_{fixed}(h) + L \cdot \sum_{s} C_{live}(s) \cdot n(s) + D \cdot \sum_{s} C_{dead}(s) \cdot n(s)$$

where n(s) is the normalized object size distribution function, i.e. n(12) is the percentage of objects with size equal to 12 bytes, L is the number of live objects and D is the number of dead objects. Figure 7 shows the accumulative object size distribution function for the test applications. Applications such as `db` and `mtrt` are dominated by one object size, whereas other applications use multiple object sizes. In general, the majority (more than 90%) of objects are small, i.e., less than 100 bytes, except for `compress`.

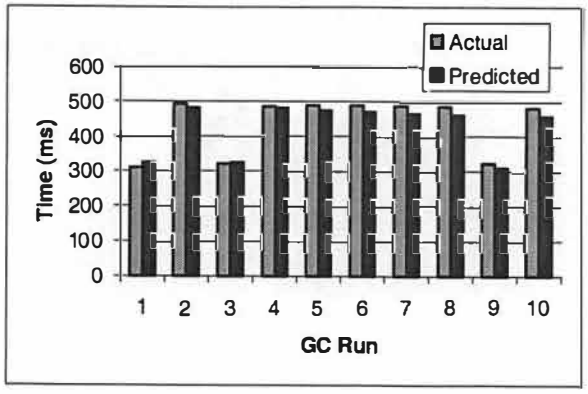
So far our formula has not taken into consideration the cost of the occasional copying performed by the collector. For our test cases, copying only occurred in two applications in four GC invocations (out of a total of

Object Size	C_{fixed} Per MB	C_{live} Per Object	C_{dead} Per Object
12	3.75	7.04E-04	3.02E-04
20	3.75	7.51E-04	3.49E-04
28	3.75	8.67E-04	4.03E-04
36	3.75	9.55E-04	4.71E-04
44	3.75	1.07E-03	5.49E-04
52	3.75	1.24E-03	6.15E-04
60	3.75	1.30E-03	6.83E-04
68	3.75	1.38E-03	6.85E-04
76	3.75	1.44E-03	6.83E-04
84	3.75	1.41E-03	6.85E-04
92	3.75	1.59E-03	6.85E-04
100	3.75	1.40E-03	6.82E-04
108	3.75	1.33E-03	6.87E-04
116	3.75	1.40E-03	6.91E-04
124	3.75	1.33E-03	6.92E-04
132	3.75	1.46E-03	7.17E-04

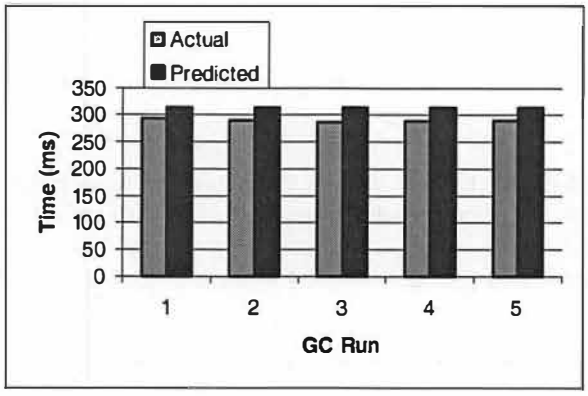
Table 4. GC Characteristics on 333MHz UltraSPARC IIi



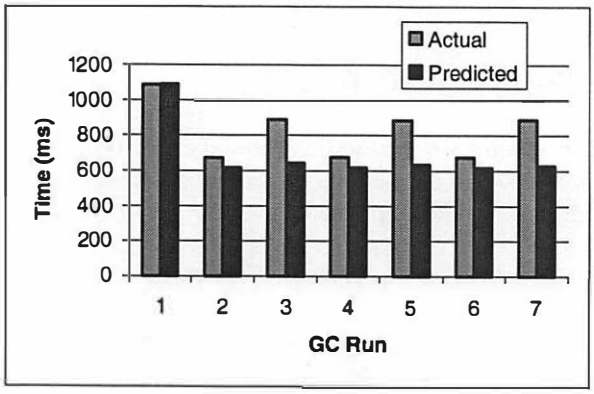
(a). _201_compress



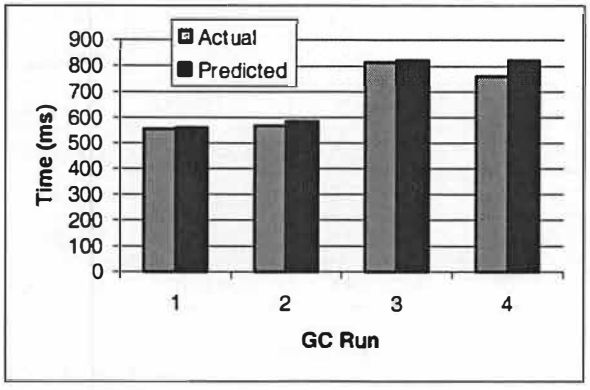
(b). _202_jess



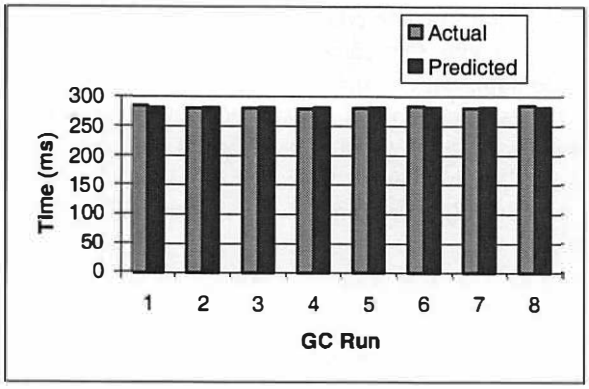
(c). _209_db



(d). _213_javac



(e). _227_mtrt



(f). _228_jack

Figure 8. Predicted versus Actual GC Times. All tests were run on the Sun SPARC workstation using a heap size of 32MB, except for javac and mtrt, which were run on a heap size of 64MB to eliminate the variation on the number of GCs from different runs.

thirty-five GC invocations). Three of those four GC invocations were explicit garbage collections made by the application, which trigger unnecessary copying. Currently we approximate this copying overhead by dividing the number of bytes copied over the memory bandwidth, and we use the actual number of bytes copied. In the future, we will enhance our analyzer to estimate this information from the application memory

characterization, assuming that the algorithm that decides when to perform a copy is known. We will also explore techniques to design microbenchmarks that would trigger a copy and measure the cost directly.

Figure 8 shows the predicted versus actual GC running times for the six SPEC applications on the Sun SPARC workstation. A summary of the percentage time difference between the predicted and the actual GC times is presented in Table 5.

For `compress` (Figure 8(a)), there are five garbage collections during the execution of the compress application. The predicted GC times match the actual times quite closely (with 0.2% error rate), showing that our prediction model works well in this case. In the fourth GC run, the collector copied certain live objects to the beginning of the heap, which accounts for the boost in the GC time. The result shows that our approximation on the copying time works well in this case also.

Figures 8(b), 8(c), 8(e) and 8(f) show the results for `jess`, `db`, `mtrt` and `jack`, respectively. The predicted times track the actual times quite closely. No copying occurred in these cases.

Figure 8(d) shows the results for `javac`. The predicted times track the actual times nicely except for the 3rd, 5th, and 7th GC runs. It turns out that these three GCs were invoked explicitly by the application at times when the heap space had not been exhausted and most objects on the heap were live objects. The explicit GCs also trigger unnecessary copying of live objects. In this case, our approximation on the copying cost does not work well. This might be due to the fact that the approximation does not include the overhead for initiating a copy, therefore it underestimates the cost in cases when many small objects are copied.

In summary, HBench:JGC is able to predict the actual GC times within 10% for five out of the six applications (Table 5). In the case of `javac`, the error rate is -6.4% if we disregard the three explicit GCs. The results demonstrate that the vector-based methodology used by Hbench:JGC is a promising technique for predicting application performance. In addition, we believe that when equipped with a better profiler and analyzer, the prediction accuracy of HBench:JGC can be improved further.

SPEC Application	Stdev (%)	Time Difference (%)
_201_compress	0.5	0.2
_202_jess	0.4	-2.2
_209_db	0.8	8.3
_213_javac	0.5	-15.8(-6.4*)
_227_mtrt	9.5	3.1
_228_jack	0.5	-0.2

* Results if we discard 3 explicit GCs.

Table 5. Summary of Predicted vs. Actual GC Times

5. Discussion and Future Work

In this section we discuss issues that might arise when using HBench:JGC on more sophisticated GC implementations such as those presented in Section 2.1.2, and how we plan to address these issues.

Concurrent garbage collection presents some technical challenges. With concurrent garbage collection, the application can continue to allocate new objects and access objects on the heap while a garbage collection is in process. Measuring the GC time is difficult because the GC time is dispersed in application execution time. We plan to approach this problem in the following way. We run a standard Java application without garbage collection, and then we run the same application with an additional thread that continuously allocates objects and invokes garbage collection. The performance degradation observed when the application is run with the additional GC intensive thread should be a good approximation of the GC time.

Many concurrent collectors are also incremental. Therefore, we will need to estimate the percentage of the heap that is scanned by the collector. In most cases, an incremental collector sets an upper bound on the number of root objects to be processed, from which one can estimate the number of objects on the heap to be scanned.

Predicting the performance of parallel garbage collectors can be potentially difficult because the speed-up of a parallel GC run over its sequential counterpart depends not only on the degree of parallelism, but also on how balanced each thread's load is and the interactions between the threads such as lock contention. Analyzing performance of multi-threaded applications in general is still an active area of research.

To apply HBench:JGC to generational garbage collectors, we model the collector performance for each generation, and then combine them together to form the total GC time. To achieve that, our profiler needs to be enhanced with the capability to estimate the object life expectancy. Furthermore, our analyzer should be able to predict when objects are promoted to older generations, i.e., it needs to know the age threshold for promotion. Some GC implementations make this knowledge public. For implementations that do not, we need to design our microbenchmark suite such that it can deduce the age threshold by creating and deleting objects at different rates.

Currently, the memory cache effect is included in our cost functions as a function of object size. Our results indicate that in some cases, this simple model might be insufficient. We are investigating ways to model the memory cache hierarchy explicitly.

Our short-term goal is to experiment on more garbage collector implementations and include more applications in our experiments. In the long run, we expect to refine our model to cope with more sophisticated GC implementations and incorporate HBench:JGC into the HBench:Java suite, in order to more accurately predict a Java application's total running time.

6. Related Work

Many researchers have studied the performance of dynamic memory management [5][15]. This literature provides a good foundation for understanding the inherent cost of dynamic storage allocation. Our approach differs in the goals we try to achieve. We emphasize predictability — the ability to predict application performance on different GC implementations without running the application on target implementations. In contrast, past research has focused on comparing the cost of memory management by running a set of popular applications on target memory management implementations.

Knuth [8] presents a comprehensive analysis and comparison of the time complexity of several dynamic storage management algorithms. This systematic approach to benchmarking memory management algorithms offers insight into the efficiency of these algorithms and helps explain the performance differences. However, the analysis assumes certain statistical properties for both memory allocation and liberation patterns and only applies when the system reaches equilibrium.

In [4], Cohen et al. compare performance of four compacting algorithms using analytical models. The analytical models are parameterized by the amount of work to be done, such as the number of cells (objects), number of pointers (links) and related information, and the time to perform the basic operations common to all compactors, such as the time to test a conditional expression. Their goal is similar to ours in that they also try to estimate GC execution times "without resorting to empirical tests". The main difference lies in the level of abstraction used for the primitive (elementary) operations. Their primitive operations are low-level machine instructions, whereas we conglomerate all machine instructions performed on an object into a single per-object operation (e.g., per-live-object overhead). Because their primitives are at such a low-level, their models are more elaborate and require intimate knowledge of the algorithms (i.e., the complete source code). Furthermore, as computer architectures become more advanced, machine-level optimizations and the memory cache hierarchy could introduce significant side effects such that the analytical model will no longer be applicable. In our case, the cost of primitives is measured explicitly by the microbenchmark and therefore includes these side effects.

7. Conclusion

HBench:JGC is a vector-based, application-specific benchmarking framework for evaluating garbage collector performance. Our results demonstrate HBench:JGC's unique predictive power. By taking the nature of target applications into account and offering fine-grained performance characterizations of garbage collectors, HBench:JGC can provide meaningful metrics that help better understand and compare GC performance.

8. Acknowledgement

Many of the ideas presented in this paper were inspired by discussions with Dave Detlefs from Sun Microsystems Labs. We would also like to thank the anonymous COOTS reviewers and our shepherd Chris Small for their valuable comments and suggestions in improving the paper.

9. Bibliography

[1] Boehm, H., and Weiser, M., "Garbage Collection in an Uncooperative Environment." *Software–Practice and Experience*, pages 807-820, September 1988.

[2] Baker, H. G. Jr., "List Processing in Real Time on a Serial Computer." *Communications of the ACM*, 21(4), pages 280-294, April 1978.

[3] Cheney, C. J., "A Non-Recursive List Compacting Algorithm." *Communications of the ACM*, 13(11), pages 677-678, November 1970.

[4] Cohen, J., and Nicolau, A., "Comparison of Compacting Algorithms for Garbage Collection." *ACM Transactions on Programming Languages and Systems*, 5(4), pages 532-553, 1983.

[5] Detlefs, D., Dosser, A., and Zorn, B., "Memory Allocation Costs in Large C and C++ Programs." *Software–Practice and Experience*, 24(6), pages 527-542, June 1994.

[6] JVMPI, Java Virtual Machine Profiling Interface. http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/index.html.

[7] Jones, R., and Lins, R. D., *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Son Ltd, New York, 1996.

[8] Knuth, D. E., *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Second Edition, Addison Wesley, Reading, MA, 1973.

[9] Larose, M., and Feeley, M., "A Compacting Incremental Collector and its Performance in a Production Quality Compiler." In *Proceedings of the 1998 International Symposium on Memory Management*, pages 1-9, Vancouver Canada, October 17-19, 1998.

[10] Lieberman, H., and Hewitt, C., "A Real-Time Garbage Collector Based on the Lifetimes of Objects." *Communications of the ACM*, 26(6), pages 419-429, June 1983.

[11] Seltzer, M., Krinsky, D., Smith, K., and Zhang X., "The Case for Application-Specific Benchmarking." In *Proceedings of the 1999 Workshop on Hot Topics in Operating Systems (HotOS VII)*, pages 102-107, Rio Rico, AZ, March 29-30, 1999.

[12] Smith, F., and Morrisett, G., "Comparing Mostly-Copying and Mark-Sweep Conservative Collection." In *Proceedings of the 1998 International Symposium on Memory Management*, pages 68-78, Vancouver Canada, October 17-19, 1998.

[13] SPECJVM98 Benchmarks. August 19, 1998 release. http://www.spec.org/osg/jvm98.

[14] Zhang, X., and Seltzer, M., "Hbench:Java – An Application-Specific Benchmarking Framework for Evaluating Java Virtual Machines." In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 62-70, San Francisco, CA, June 3-4, 2000.

[15] Zorn, B., "The Measured Cost of Conservative Garbage Collection." *Software–Practice and Experience*, 23(7), pages 733-756, July 1993.

Distributed Garbage Collection for Wide Area Replicated Memory

Alfonso Sánchez Luís Veiga Paulo Ferreira
INESC/IST,
Rua Alves Redol Nº 9,
Lisboa 1000-029, Portugal
{alfonso.sanchez, luis.veiga, paulo.ferreira}@inesc.pt

Abstract

It is well known that distributed systems pose serious difficulties concerning memory management: when done manually, it leads to memory leaks and dangling references causing applications to fail. We address this problem by presenting a distributed garbage collection (DGC) algorithm for distributed systems supporting replicated data over wide area networks.

Current DGC algorithms are not well suited for such systems because either (i) they do not consider the existence of replication, or (ii) they impose severe constraints on scalability by requiring causal delivery to be provided by the underlying communication layer.

Our algorithm solves these problems by (i) adapting classical reference-counting DGC algorithms that were conceived for non-replicated systems (e.g. indirect reference-counting, SSP chains, etc.), and (ii) improving our previous algorithm for replicated systems (i.e. Larchant).

The result is a DGC algorithm that, besides being correct in presence of replicated data and independent of the protocol that maintains such replicas coherent among processes, it does not require causal delivery to be ensured by the underlying communications support. In addition, it has minimal performance impact on applications.

1 Introduction

Modern distributed applications sharing long-term data over many places, geographically separated, appear each day. Typical examples are found in the fields of concurrent engineering, cooperative applications, etc.

Manual memory management is extremely difficult when developing the aforementioned distributed applications. The reason is that graphs of reachability are large, widely distributed and frequently modified through assignment operations executed by applications. In addition, data replicated in many processes is not necessarily coherent making manual memory management much harder. For these reasons it is impossible to do manual memory management without generating dangling references and/or memory leaks.

Automatic memory management, also known as Garbage Collection (GC), is the single realistic option which is able to maintain referential integrity (i.e. no dangling references or memory leaks) in Wide Area Replicated Memory (WARM) systems. As a result, program reliability and programmer productivity are clearly improved.

1.1 Shortcomings of Current Solutions

Current DGC algorithms [1, 15] are not well suited for WARM systems based on data-shipping because of the following drawbacks: either (i) they do not consider the existence of replication, or (ii) they impose severe constraints on scalability by requiring causal delivery to be supported by the underlying communication layer.

The first drawback, i.e. not considering replicated data, concerns all the classical DGC algorithms that were designed for function-shipping based systems, such as Indirect Reference Counting (IRC) [14] or SSP Chains [16]. As a matter of fact, these algorithms are not safe in presence of replicated data, as explained now.

Consider Figure 1 in which an object x is replicated in processes i and j; each replica of x is noted x_i and x_j, respectively. Now, suppose that x_i contains a reference to an object z in another process k, x_j points to no other object, x_i is locally unreachable and x_j is locally reachable[1]. Then, the question is:

[1]Locally (un)reachability is related to (un)accessibility

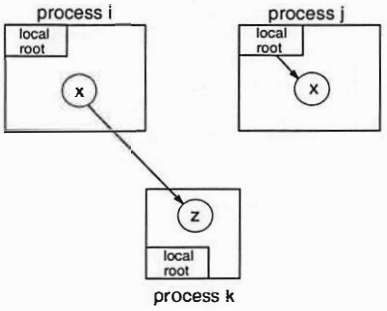


Figure 1: *Safety problem of current DGC algorithms which do not handle replicated data: z is erroneously considered unreachable.*

should z be considered garbage? Classical DGC algorithms consider that z is effectively garbage. However, this is wrong because, in a WARM system, it is possible for an application in j to "acquire" a replica of x from some other process, in particular, x_i[2]. Thus, the fact that x_i is *locally* unreachable in process i does not mean that x is *globally* unreachable; as a matter of fact, x_i contents can be accessed by an application in process j by means of an "acquire". Therefore, in a WARM system, a target object z is considered unreachable only if the union of all the replicas of the source object, x in this example, do not refer to it. We call this the Union Rule (more details in Section 4.2.2).

The second drawback, i.e. imposing severe constraints on scalability, affects current DGC algorithms conceived for WARM systems, such as Larchant [5, 10]. As a matter of fact, such algorithms are not scalable because they require the underlying communication layer to support causal delivery.

So, in conclusion, classical DGC algorithms, such as IRC and SSP Chains, are not safe for WARM systems but promise to be scalable, in particular, do not require causal delivery; on the other hand, WARM specific DGC algorithms, such as Larchant, deals safely with replication but lacks scalability.

Thus, the main contribution of this work is the following: showing how classical DGC algorithms (conceived for function-shipping based systems) can be extended to handle replication while keeping their scalability.

We do not address the issue of fault-tolerance, i.e. it is out of the scope of the paper how the algorithm behaves in presence of communication failures and processes crashes. However, solutions similar to those found in classical DGC algorithms can also be applied (for example, leasings as in RMI [18].

This paper is organized as follows. In Section 2 we present the model of a WARM for which the DGC was defined. The DGC algorithm is described in Sections 3 and 4. Section 5 highlights some of the most important implementation aspects. Section 6 presents some performance results from a real application. The paper ends with some related work and conclusions in Section 7 and 8, respectively.

2 WARM Model

This section presents the model for Wide Area Replicated Memory (WARM). A WARM is a replicated distributed memory spanning several processes. These processes are connected in a network and communicate only by asynchronous message passing. We indicate that a message M has been sent from process i to process j as $\langle send.M \rangle_{i \to j}$; the delivery of that message is noted $\langle deliver.M \rangle_{i \to j}$.

In a WARM, the only way to share information is by replication of data, which can be done with a DSM based mechanism[12]. Thus, processes do not use Remote Procedure Call (RPC) to access remote data.

It's worthy to note that application code inside a process never sends messages explicitly. Instead, application code access data always locally; transparently to the application code, the WARM runtime system is responsible to replicate data locally when needed.

Each participating process in the WARM encloses, at least, the following entities: memory, mutator[3], and a coherence engine. In our WARM model, for each one of these entities, we consider only the operations that are relevant for GC purposes.

We believe that our model is sufficiently general to describe most distributed systems supporting wide area applications using data shipping. This model clearly defines the environment for which the DGC algorithm is conceived.

from the enclosing process's local root.

[2] In distributed systems with replicated data, an "acquire" operation allows a process to update its local replica of a particular object with the contents of another replica, of that same object, residing in some other process with a data-shipping mechanism.

[3] The term mutator [7] designates the application code which, from the point of view of the garbage collector, *mutates* (or modifies) the reachability graph of objects.

2.1 Memory Organization

An **object** is defined to be a consecutive sequence of bytes in memory. Applications can have different views of objects and can *see* them as language-level class instances, memory pages, data base records, web pages, etc.

Objects can contain **references** pointing to other objects. An **outgoing inter-process** reference is a reference to a target object in a different process. An **incoming inter-process** reference is a reference to an object that is pointed from a different process. Our model does not restrict how references are actually implemented. They can be virtual memory pointers, URLs, etc.

An object is said to be **reachable** if it is attainable directly or indirectly from a GC root (defined in Section 3.1). An object is said to be **unreachable** if there is no reference path (direct or indirect) from a GC root leading to that object.

The unit for coherence is the object. Any object can be replicated (i.e. cached) in any process. A replica of object x in process i is noted x_i. Each process can cache a replica of any object for reading or writing according to the coherence protocol being used.

2.2 Mutator model

The single operation executed by mutators, which is relevant for GC purposes, is **reference assignment**; this is the only way for applications to modify the graph of objects.

The reference assignment operation executed by a mutator in some process i is noted $<x := y>_i$. This means that a reference contained in object x is assigned to the value of a reference contained in object y.[4] This assignment operation results in the creation of a new inter-process reference from x to z, as illustrated in Figure 2.

Obviously, other assignments can delete references transforming objects in garbage. For example, in Figure 2, if the mutators in processes i and j perform $<x := 0>_i$ and $<y := 0>_i$, object z becomes unreachable, i.e. garbage, given that there are no references pointing to it.

In conclusion, assignment operations (done by mutators) modify the object graph either creating or deleting references. An object becomes unreachable when the last reference to it disappears; when this occurs, such an object can be safely reclaimed by the garbage collector because there is no possibility for any process to access it.

[4] This notation is not fully accurate but it simplifies the explanation of the DGC algorithm. As a matter of fact, to be more precise we should write x.ref = y.ref (C++ style notation). However, this improved precision is not important for the DGC algorithm description and would complicate it un-necessarily.

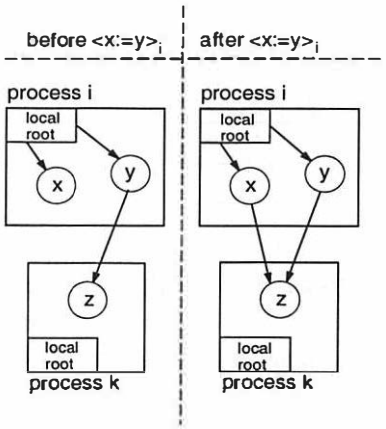


Figure 2: *Creation of a new inter-process reference to object z through an assignment operation.*

2.3 Coherence Model

The coherence engine is the entity of the WARM that is responsible to manage the coherence of replicas. The coherence protocol effectively used varies from system to system and depends on several factors such as the number of replicas, distances between processes, and others. However, the only coherence operation, which is relevant for GC purposes, is the **propagation** of an object, i.e. the replication of an object from one process to another. The propagation of an object y from process i to process j is noted $\text{propagate}(y)_{i \to j}$.

We assume that any process can propagate a replica into itself as long as the mutator causing the propagation holds a reference to the object being propagated. Thus, if an object x is locally unreachable in process i, the mutator in that process can not force the propagation of x to some other process; however, if some other process j holds a reference to x, it can request x to be propagated from i to j (as occurs in Figure 1).

We assume that, in each process, the coherence engine holds two data structures, called **inPropList** and **outPropList**; these indicate the process *from which* each object has been propagated, and the processes *to which* each object has been propagated, respectively [5]. Thus, each entry of the inPro-

[5] Usually, this information does exist in the coherence engine in order to manage the replicas.

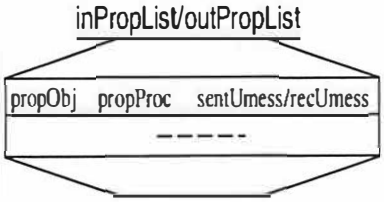


Figure 3: inPropList *and* outPropList *internal data.*

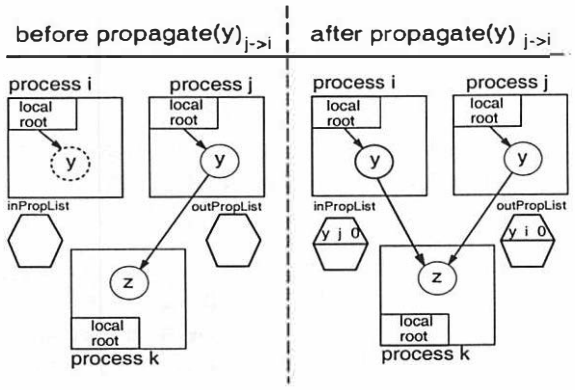


Figure 4: *Coherence engine propagates object* y *from process* j *to process* i. *The dashed line of* y_i *means that initially, in process* i, y *is not yet replicated in* i.

pList/outPropList contains the following information (see Figure 3):

- **propObj** - the reference of the object that has been propagated into/to a process;
- **propProc** - the process from/to which the object propObj has been propagated;
- **sentUmess/recUmess** - bit indicating if a unreachable message (more details in Section 3.2) has been sent/received.

When an object is propagated to a process we say that its enclosed references are **exported** from the sending process to the receiving process; on the receiving process, i.e. the one receiving the propagated object, we say that the object references are **imported**.

Figure 4 illustrates the effect of a propagation. Object z has no replicas. Initially, only process j caches a replica of y; thus, both outPropList and inPropList of processes j and i are empty. In addition, y_j points to z. After y has been replicated from process j to process i, a new inter-process reference from y_i to z is created; this is due to the fact that the reference to z was exported from process j to (be imported by) process i. The inPropList and outPropList reflect this situation.

In order to understand how the DGC algorithm works it is important to emphasize the following aspects concerning the creation of inter-process references. The only way a process can create an inter-process reference is through the execution of only two operations: (i) reference assignment, which is performed explicitly by the mutator, and (ii) object propagation, which is performed by the coherence engine in order to allow the mutator to access some object[6].

3 Distributed Garbage Collection Algorithm

In this section we describe the DGC algorithm and its data structures. Then, in Section 4 we go into more detail by describing a prototypical example which addresses all the aspects of the DGC algorithm.

The DGC algorithm is an hybrid of tracing and reference-counting. Thus, each process has two GC components: a local tracing collector, and a distributed collector. Each process does its local tracing independently from any other process. The local tracing can be done by any mark-and-sweep based collector. The distributed collectors, based on reference-counting, work together by changing asynchronous messages, as described in the following sections. In the rest of the paper we focus on distributed collection.

3.1 Data Structures

A **stub** describes an outgoing inter-process reference, from a source process to a target process. A **scion** describes an incoming inter-process reference, from a source process to a target process. It is important to note that stubs and scions do not impose any indirection on the native reference mechanism. In other words, they do not interfere either with the structure of references or the invocation mechanism. They are simply GC specific auxiliary data structures.

A stub stores in its internal data structures the following information:

- **OutRef** - the reference of the target object;

[6]For example, in some DSM-based systems, when the mutator tries to access an object that is not yet cached locally, a page fault is generated; then, this fault is automatically recovered by the coherence engine that obtains a replica of the faulted object from some other process.

- **SourceObj** - the reference of the local object containing the outgoing inter-process reference;
- **Scion** - the identification of the corresponding scion; and
- **Chain** - the identification of a stub or a scion in the same process.

A scion stores in its internal data structures the following information:

- **InRef** - the reference of the target object;
- **Stub** - the identification of the corresponding stub; and
- **Chain** - the identification of a stub or a scion in the same process.

Finally, a **process's GC root** includes: (i) the **local root**, i.e. stacks and static variables, (ii) the set of scions of that process, and (iii) the lists inPropList and outPropList.

3.2 Algorithm

The local and distributed collectors depend on each other to perform their job in the following way. A local collector running inside a process traces the object graph locally cached; the starting point of the trace is the process's GC root. A local tracing generates a new set of stubs; it is based on this new set that the distributed collector, in that process, may decide to update remote scions in other processes.

3.2.1 Local Collector

The local collector starts the graph tracing from the process's local root and set of scions. For each outgoing inter-process reference it creates a stub in the new set of stubs. Once this tracing is completed, every object locally reachable by the mutator has been found (e.g. marked, if a mark-and-sweep algorithm is used); objects not yet found are locally unreachable; however, they can still be reachable from some other process holding a replica of, at least, one of such objects (as is the case of x_i in Figure 1). To prevent the erroneous deletion of such objects, the collector traces the objects graph from the lists inPropList and outPropList, and performs as follows.

- When a locally reachable object (previously discovered by the local collector) is found, the tracing along that reference path ends.
- When an outgoing inter-process reference is found the corresponding stub is created in the new set of stubs.
- For an object which is reachable only from the inPropList, a message unreachable is sent to the site from where that object has been propagated; this sending event is registered by changing a sentUmess bit in the corresponding inPropList entry from 0 to 1.[7]

  When a unreachable message reaches a process, this delivery event is registered by changing a recUmess bit in the corresponding outPropList entry from 0 to 1.
- For an object which is reachable only from the outPropList, and the enclosing process has already received a unreachable message from all the processes to which that object has been previously propagated, a reclaim message is sent to all those processes and the corresponding entries in the outPropList are deleted; otherwise, nothing is done.

  When a process receives a reclaim message it deletes the corresponding entry in the inPropList.

3.2.2 Distributed Collector

The main ideas behind the DGC algorithm can be summarized as follows.

- As already mentioned, an object can be reclaimed only when all its replicas are no longer reachable. This is ensured by tracing the objects graph from the lists inPropList and outPropList; objects that are reachable only from these lists are not locally reachable (i.e. by the local mutator); however, they can not be reclaimed without ensuring their global unreachability, i.e. that none of their replicas are accessible. This will be explained in detail in the following section.
- The DGC algorithm is independent of the particular coherence protocol implemented by the coherence engine. In other words, the DGC algorithm does not require waiting for replicas to be coherent.

[7] Note that from now on, the replica is not reachable by the local mutator; if another propagate operation occurs bringing a *new* replica of that same object into the process, the *old* replica remains locally unreachable, and a new entry is created in the inPropList with the corresponding sentUmess set to 0.

message	sent/received by	sent when
unreachable	LGC/DGC	object replica is reachable only from the inPropList
reclaim	LGC/DGC	all object replicas are reachable only from the inPropLists
newSetStubs	DGC/DGC	a new set of stubs is available

Table 1: *GC related* messages.

- Whatever the coherence protocol, there is only one interaction with the DGC algorithm. This interaction is twofold: (i) immediately before a propagate message is sent, the references being *exported* (contained in the propagated object) must be found in order to create the corresponding scions, and (ii) immediately before a propagate message is delivered, the outgoing inter-process references being *imported* must be found in order to create the corresponding local stubs, if they do not exist yet.[8]
- From time to time, possibly after a local collection, the distributed collector sends a message called newSetStubs; this message contains the new set of stubs that resulted from the local collection; this message is sent to the processes holding the scions corresponding to the stubs in the previous stub set. In each of the receiving processes, the distributed collector matches the just received set of stubs with its set of scions; those scions that no longer have the corresponding stub, are deleted.
- As previously described, when a local collection takes place two kinds of messages may be sent: unreachable and reclaim. On the receiving process, these messages are handled by the distributed collector that performs the following operations: sets the recUmess bit in the corresponding outPropList entry, and deletes the corresponding entry in the inPropList, respectively.
- The DGC algorithm does not require the underlying communication layer to support causal delivery.

Table 1 presents all the GC related messages of the model, the components responsible for sending and receiving them, and when they occur. In Table 2 we present all the events with impact on the GC and the corresponding actions taken. These two tables summarize the way GC is performed. In the next section we describe the DGC algorithm in more detail using a prototypical example.

[8]Note that this may result in the creation of chains of stub-scion pairs, as it happens in the SSP Chains algorithm [16].

4 Prototypical Example

We use a prototypical example, illustrated in Figures 5 and 6. This example evolves along a sequence of steps covering all the situations, relevant for GC, that occur in a WARM: (i) creation of a new outgoing inter-process reference by means of a propagate operation, (ii) creation of a new outgoing inter-process reference by means of an assignment operation, and (iii) deletion of outgoing inter-process references by means of assignment and propagate operations. We show how all these occurrences affect the GC specific data structures and messages.

In the initial situation both x and y are cached in processes i and j. However, only the replica y_j points to object z in process k. There is a single stub-scion pair (s2-s1) describing the only outgoing inter-process reference from y_j to z. For the sake of simplicity of our description, we assume that this stub-scion pair is created when the system boots.[9]

Then, the sequence of steps of the prototypical example considers the following operations (see Figures 5 and 6; the effects of the operations are shown in bold).

Step 1 - Propagate y from process j to process i; this results in the creation of a new outgoing inter-process reference from object y in i to object z in k.

Step 2 - The operation $<x := y>_i$ is performed by the mutator in i; this creates a new outgoing inter-process reference from object x in i to object z in k.

Step 3 - Propagate x from process i to process j; this results in the creation of a new outgoing inter-process reference from object x in j to object z in k.

Step 4 - The operation $<y := 0>_j$ is performed by the mutator in j; this results in the deletion of an outgoing inter-process reference from object y in j to object z in k.

[9]For example, the reference to z could be obtained from a name service.

event	occurs when	action taken
reference exported	propagate an object from a process	create scion
reference imported	propagate an object into a process	create stub
object replica reachable only from the inPropList	LGC runs	send unreachable message to the process with the corresponding outPropList entry; set the sentUmess bit accordingly
unreachable message received	unreachable message sent	set the recUmess bit accordingly; if all recUmess bits for a particular object are set, then send the corresponding reclaim messages and delete the outPropList entry
reclaim message received	reclaim message sent	delete corresponding inPropList entry
new set of stubs available	LGC runs	newSetStubs message sent to the processes holding the scions corresponding to the previous set of stubs
newSetStubs message received	newSetStubs message sent	compare stubs set with set of scions; delete scions with no corresponding stubs

Table 2: *GC related events.*

Step 5 - Propagate y from process j to process i; this results in the deletion of an outgoing inter-process reference from object y in i to object z in k.

Step 6 - The operation $<x := 0>_i$ is performed by the mutator in i; this results in the deletion of an outgoing inter-process reference from object x in i to object z in k.

Step 7 - the mutator in j deletes the reference from the local root to object x.

Step 8 - the mutator makes x_i unreachable by deleting the reference from the local root; thus, every replica of x becomes garbage.

The prototypical example presented above has two parts: the first three steps results in the creation of new outgoing inter-process references; the last five steps result in z becoming unreachable. In the next sections we describe how the DGC works in order to deal with this prototypical example.

4.1 Creation of Outgoing Inter-process References

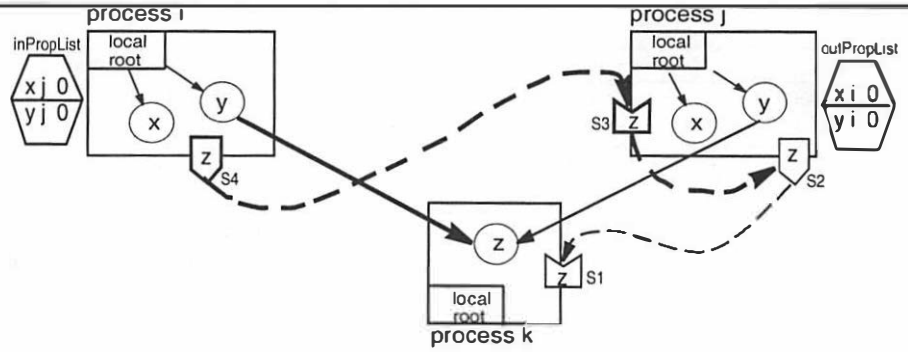
In the prototypical example, the creation of outgoing inter-process references occurs first by propagation (step 1), then by reference assignment (step 2), and finally by propagation again (step 3). We address these cases now.

4.1.1 Propagation

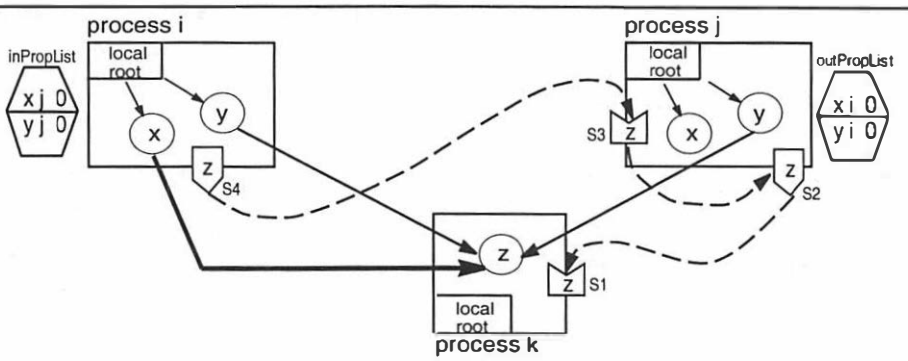
The first operation in the prototypical example is $propagate(y)_{j \rightarrow i}$ (Figure 5, step 1). Immediately before this message is sent from process j, object y must be scanned for references being exported. For each one of these references, the corresponding scion must be created. In this case, y contains only one reference (pointing to z); the corresponding scion **s3** is shown in bold. Note that the scion just created, through its **Chain** field, refers to the already existing stub **s2** (describing the outgoing inter-process reference from object y to object z).

Immediately before $propagate(y)_{j \rightarrow i}$ is delivered in process i, object y has to be scanned for imported outgoing inter-process references in order to create the corresponding stubs in process i, if they do not exist yet. In the prototypical example, y contains a single reference and there is no stub describing it in process i. Thus, the corresponding stub **s4** is created (shown in bold); this stub, through its internal data structures, refers to the scion previously created in process j. Then, the mutator may freely access object y in process i.

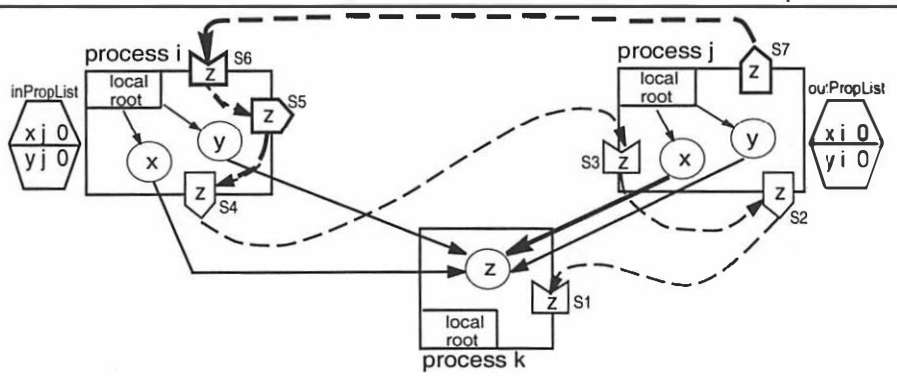
Thus, the information stored in the stub-scion



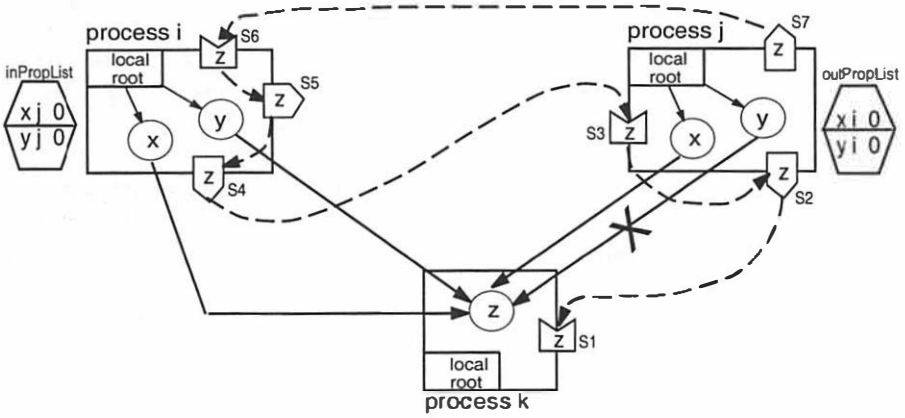
Step 1: propagation of object y from process j to process i.



Step 2: creating a new inter-process reference through $<x:=y>_i$.

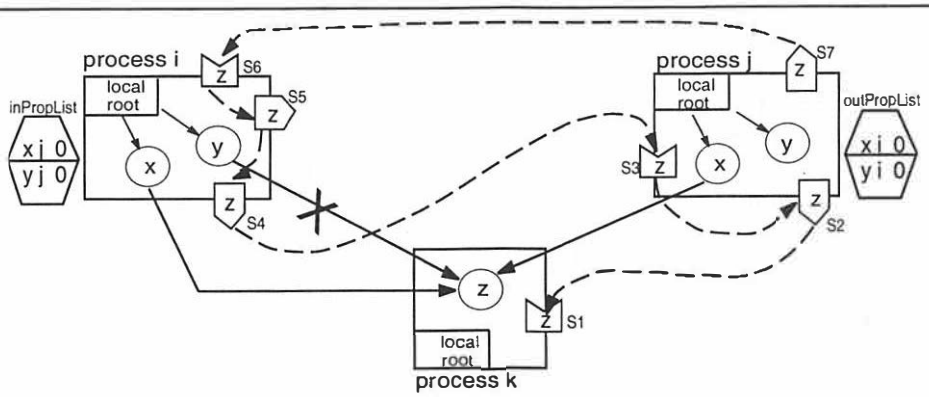


Step 3: propagation of object x from process i to process j.

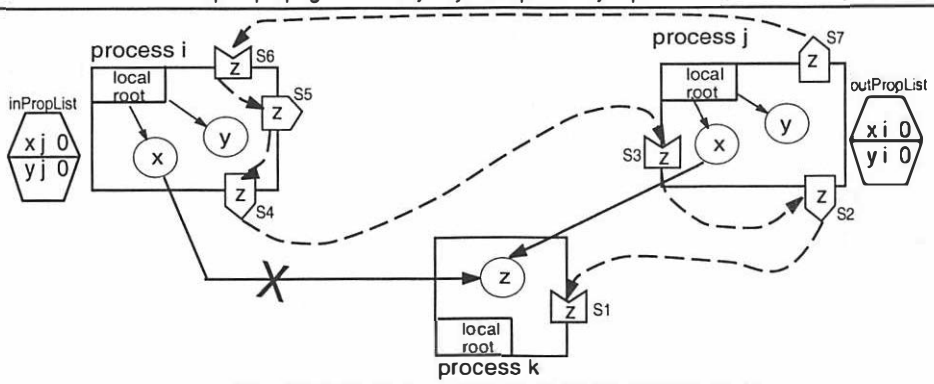


Step 4:deleting an inter-process reference through $<y:=0>_j$.

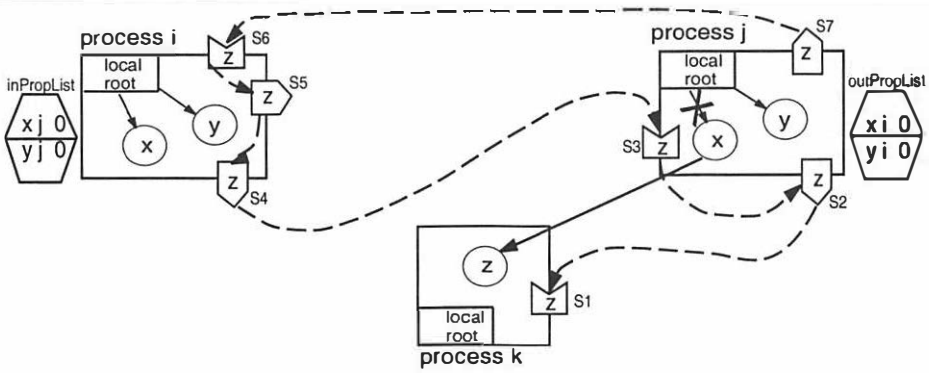
Figure 5: *Prototypical example (part 1).*



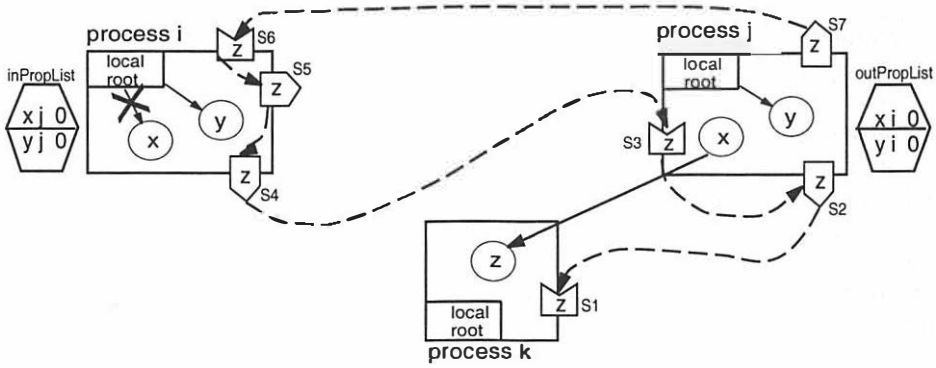
Step 5: propagation of object y from process j to process i.



Step 6:deleting an inter-process reference through $<x:=0>_i$.



Step 7:object x in process j becomes unreachable from the local root.



Step 8:object x in process i becomes unreachable from the local root.

Figure 6: *Prototypical example (part 2).*

pair just created, s4-s3, is the following:

- stub s4: OutRef refers to object z in process k, sourceObj refers to object y in process i, Scion identifies the corresponding scion s3 previously created in process j, and Chain is null;
- scion s3: InRef refers to object z in process k, Stub identifies the corresponding stub s4 in process i, and Chain refers to the stub describing the outgoing inter-process reference from object y to object z.

It is worthy to note that the mutator does not have to be blocked while the GC specific operations mentioned above are executed (scanning the object being propagated and creating the corresponding scion and stub); such operations can be executed in the background.

To summarize, there are the following rules:

> **Safety Rule I: Clean Before Send Propagate.** *Before sending a propagate message for an object* y *from a process* j, y *must be cleaned (i.e. it must be scanned for references) and the corresponding scions created in* j.

> **Safety Rule II: Clean Before Deliver Propagate.** *Before delivering a propagate message for an object* y *in a process* i, y *must be cleaned (i.e. it must be scanned for outgoing inter-process references) and the corresponding stubs created in* i, *if they do not exist yet.*

4.1.2 Assignment

The second step of the prototypical example is the execution of the operation $<x := y>_i$. This results in the creation of a new outgoing inter-process reference: from object x in process i to z in process k. There is absolutely no operation to be done on behalf of the DGC algorithm.

This might seem strange because, according to traditional reference counting algorithms [20], each time a reference is created, a counter (at least) must be incremented. In a WARM, where mutators may create inter-process references very easily and frequently, through a simple reference assignment operation, such increment would be extremely inefficient. As a matter of fact, this would require instrumenting every reference assignment and increment a counter accordingly, possibly on some remote process. In the following sections it will become clear that such increment (or equivalent operation) does not need to be performed immediately.

4.1.3 Propagation

The third step of the prototypical example is the propagation of object x from process i to process j. This results in the creation of a new outgoing inter-process reference: from object x in process j to object z in process k(shown in bold in Figure 5, step 3).

According to Safety Rule **Clean Before Send Propagate**, before the propagate message is sent, the following has to be done in process i: scan object x, find its enclosed references and create the corresponding scions. In this case, object x has only one reference; thus, as a result of the scan, scion s6 is created in process i (shown in bold, Figure 5, step 3).

In addition, it is created stub s5 describing the outgoing inter-process reference from object x in process i to object z in process k.[10]

According to Safety Rule **Clean Before Deliver Propagate**, before the propagate message is delivered in process j, object x must be cleaned and the corresponding stub s7 created (shown in bold, Figure 5, step 3).

4.2 Deletion of Outgoing Inter-process References

In the prototypical example, the deletion of outgoing inter-process references occurs first by reference assignment, then by propagation, then by reference assignment again, and finally by propagation again. After all these operations, object z is unreachable. We address these steps now.

4.2.1 Assignments and Propagations

The fourth step of the prototypical example is the execution of the operation $<y := 0>_j$. This results in the deletion of the outgoing inter-process reference, from object y to object z (Figure 5, step 4). At this moment, there is absolutely no operation to be done for GC purposes.

The fifth step of the prototypical example is $\text{propagate}(y)_{j \to i}$. Given that the replica that is being propagated to i no longer points to any object, after the propagate is delivered, the outgoing inter-process reference from object y in process i to z, is (implicitly) deleted (Figure 6, step 5). At this moment, there is absolutely no operation to be done for GC purposes. Note that, given that the object being propagated contains no references, both safety

[10] Note that if a local collection has previously taken place in process i, stub s5 would have been already created.

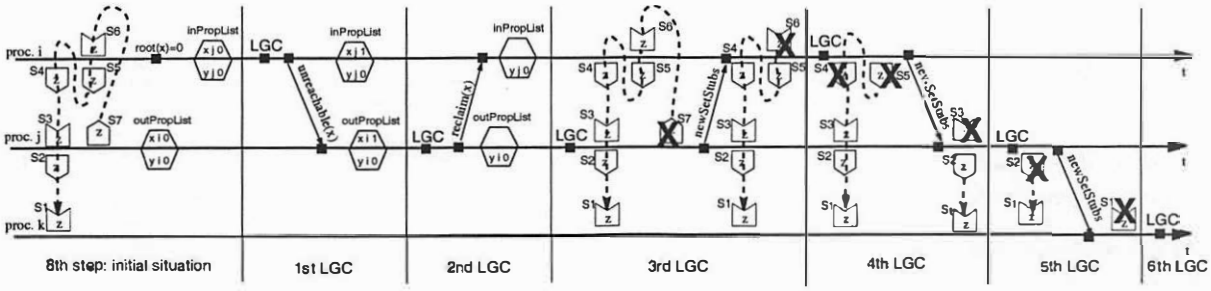


Figure 7: *Timeline describing the GC operations after the 8th step of the prototypical example.*

rules do not imply the execution of any particular operation.

The sixth step of the prototypical example is the execution of the operation <x := 0>$_i$. This results in the deletion of the outgoing inter-process reference, from object x in process i to object z in process k (Figure 6, step 6). At this moment, there is absolutely no operation to be done for GC purposes.

The seventh step of the prototypical example makes object x in process j unreachable from the local root. The last step makes object x in process i unreachable from the local root. In both cases there is absolutely no operation to be done for GC purposes.

So far, the DGC has performed no operation. In particular, no scion has been deleted. Consequently, object z, which is no longer reachable, has not been reclaimed yet. This will happen only after its protecting scion s1 in process k is deleted and the local collector is executed. Now we address the modification and deletion of stubs and scions.

4.2.2 Collecting Garbage

In step 8 of the prototypical example we see that object z will be reclaimed by the local collector in process k only after its protecting scion s1 has been deleted. This scion will be deleted only after the corresponding stub s2 in process j has disappeared; this will occur only after all the chain of stub-scion pairs s7-...-s3 gets deleted.

According to Section 3.2, the stubs and scions will disappear as a result of the local and distributed collectors in processes i and j, as explained now (see Figure 7).

1st LGC - The local collector in process i detects that object x is reachable only from the inPropList; thus, a message unreachable is sent to process j and the corresponding sentUmess bit is set.

When this message is delivered in process j, the recUmess bit in the corresponding entry of outPropList is set.

2nd LGC - The local collector in process j detects that object x is reachable only from the outPropList and the corresponding entry has its recUmess bit set to one; thus a message reclaim is sent to process i and the entry in the outPropList is deleted.

When this message is delivered in process i, the corresponding entry in inPropList is deleted.

3rd LGC - As a result of a local collection in process j, x is reclaimed and, consequently, stub s7 describing its outgoing inter-process reference to object z is not in the new set of stubs. This new set of stubs is sent as a newSetStubs message from process j to process i; then, the distributed collector in i deletes the corresponding scion s6.

Note that stub s2, in spite of the fact that y in j holds no outgoing inter-process reference anymore, is still in the new set of stubs because is reachable from scion s3 through its Chain data structure.

4th LGC - As a result of a local collection in process i, object x is reclaimed and the new set of stubs does not contain any stub (s5 and s4, in particular) because there are no outgoing inter-process references.

This new set of stubs is sent as a newSetStubs message from process i to process j; then, the distributed collector in j deletes the corresponding scion s3.

5th LGC - As a result of a local collection in process j a new set of stubs is generated in which there is no stub (i.e. s2) because there are no outgoing inter-process references.

This new set of stubs is sent as a newSetStubs message from process j to process k; then, the distributed collector in k deletes the corresponding scion s1.

6th LGC - Finally, a local collection occurs in process k and object z is reclaimed.

In conclusion, we have the following rule for replicated objects:

> **Safety Rule III: Union Rule.** *A target object z is considered unreachable only if the union of all the replicas of the source objects do not refer to it.*

In the prototypical example the objects pointing to z were the replicas of x and y. From Figure 7 it is clear that the union rule is respected. In addition, it is clear that there is no need for causal delivery to be ensured by the communication layer.

5 Implementation

We implemented our WARM distributed and local garbage collectors within a system called News Gathering (NG). In this section we briefly describe the NG application; then, we focus on the most important implementation aspects of the DGC: how the safety rules are implemented, and the stub/scion data structures.

5.1 NG Application

NG is a web-based client-server application that we developed, to support the sharing of files over the web by means of replication [19]. From the user point of view the client side of NG is a normal web browser with an extra menu button called "make-replica". This function allows the user to propagate a file into his machine, i.e., to create a local replica of the file he is looking at. Once replicated, the file can be freely accessed with any other application (possibly making the replicas to diverge). Later, this replica can be propagated back to the site from where it came from by means of a make-replica operation performed by other user running on that site. (Figure 8 illustrates the general architecture of this application.)

With NG, a typical user in site S1 browses the web (web servers supporting the NG application) and makes-replicas of some of the pages from, for example, the S2 site. These pages are then edited by the user and, once ready, are made available from

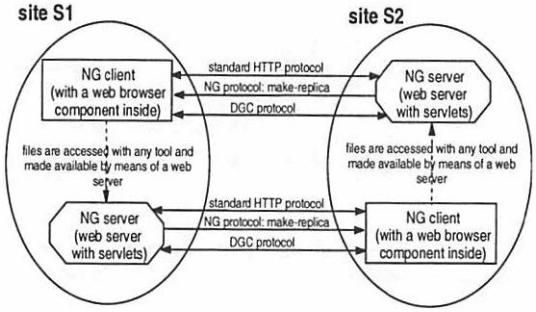


Figure 8: *General architecture of the NG application. Obviously, any number of sites is supported and not all are forced to have both a client and a server, i.e. some can be just clients or servers.*

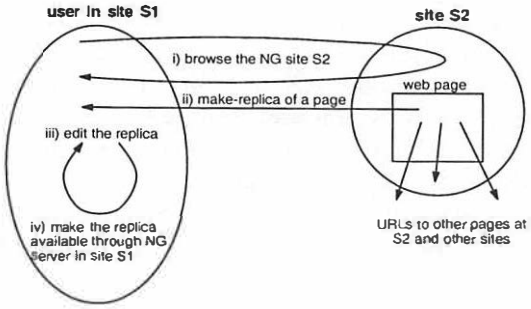


Figure 9: *Example of NG usage: i) browse the S2 web site, ii) make-replicas of a page, iii) edit the replica, and iv) make the replica available for others.*

the user's local NG server. These replicas may hold references to other (not locally replicated) S2 pages. Thus, it is desirable that such pages in the S2 web site remain available as long as there are references pointing to them. Figure 9 illustrates this scenario.

The NG application, due to the WARM distributed garbage collector, ensures that such pages at the S2 site remain there as long as they are pointed from some other NG site. In addition, files at the S2 site, which are no longer referenced from any other NG site are automatically deleted by the garbage collector. This means that neither dangling references nor memory leaks occur.

The NG application is implemented in Java; this includes the client code (that uses the Microsoft Internet Explorer component) and the servlets running within an Apache web server.

5.2 Distributed Garbage Collector

All the code of the local and distributed collectors is written in Java. The local collector is implemented as a stand-alone application. The distributed collector is implemented by the servlets and by the client.

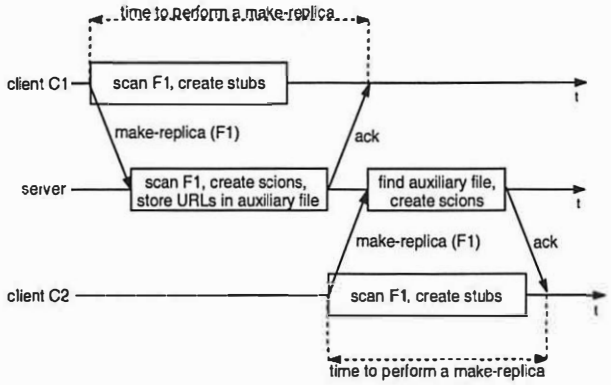


Figure 10: *Propagation of file F1.*

Basically, the code in the servlets implements the safety rule Clean Before Send Propagate (applied when a make-replica is requested); the client code implements the safety rule Clean Before Deliver Propagate (applied when the reply to a make-replica request is received). The implementation of these rules consists on scanning the web pages being propagated and creating the corresponding scions (at the server) and stubs (at the client).

The first time a file is propagated, at the server site its contents are scanned, the corresponding scions created, and the enclosed set of URLs is kept in an auxiliary file. Later, if this same page is propagated again, at the server site it only has to be scanned again if it has been modified after the last scan. The timeline presented in Figure 10 shows how the scanning needed to enforce safety rules I and II relates to the make-replica request of file F1 [11].

Another important aspect concerning the implementation of the garbage collectors (both local and distributed) is the data structures supporting the stubs and scions. These were conceived taking into account their use, in particular, to optimize the kind of information exchanged between sites that occurs when a `newSetStubs` message is sent.

This message implies that the new set of stubs, resulting from a local collection, is sent to the processes holding the scions corresponding to the stubs in the previous stub set. Then, in each of the receiving processes, the distributed collector matches the just received set of stubs with its set of scions; those scions that no longer have the corresponding stub, are deleted.

Thus, stubs are grouped by site, i.e. there is one hash table for each site holding scions corresponding to the stubs in that table. Sending a new set of stubs to a particular site is just a matter of sending the new hash table. The same reasoning applies to scions: they are stored in hash tables, each table grouping the scions whose corresponding stubs are in the same site.

6 Performance

In this section we present the most relevant performance results concerning the DGC. The critical performance results are those related to the implementation of safety rules I and II.

Thus, we downloaded a well-known web site (cnn.com) and ran on each file the code implementing the safety rules. All results were obtained in a local 100 Mbits network, connecting PCs with Windows NT, with 64 Mb of memory and a Pentium II at 233 MHz.

We downloaded all the 155 HTML files of the cnn.com web site[12] and obtained for each one the time it takes to: scan it, create the corresponding stubs, and serialize the hash table (including writing to disk). In this section, for clarity, we simply refer to the time it takes to create stubs and their size because the same values apply to scions.

file size	number of URLs	scan time	stub creation time	hash table size	time to serialize
43563	326	38	3	19252	67

Table 3: *Mean values obtained with all the files automatically downloaded from the cnn.com site (Sizes in bytes and times in milliseconds.).*

In Table 3 we present, for each one of the 155 files: the mean file size, the mean number of URLs enclosed in each file, the mean time to scan a file, the mean time it takes to create a stub in the corresponding hash table, the mean size of the hash table containing all the stubs corresponding to all the URLs enclosed in a file (that depends on the size of the corresponding URL), and the mean time

[11] Note that the client can scan the page immediately after sending a make-replica request because its contents are already available locally (for browsing).

[12] Using an automatic tool called WebReaper available from http://www.otway.com/webreaper configured with a depth level of 5.

file name	file size	number of URLs	scan time	stub creation time	hash table size	URLs size	time to serialize
europe.htm	49055	493	36	10	25485	22367	60
health.htm	102933	491	45	10	26268	23465	60
law.htm	79460	523	117	10	31373	30194	70
main.htm	67081	588	40	10	38548	34108	71
politics.htm	59079	470	90	10	25963	22939	60
showbiz.htm	71579	498	40	10	26481	24944	111
space.htm	58488	478	78	50	24835	23614	50
sports.htm	41778	366	27	10	23308	18908	60
tech.htm	49645	462	34	10	21820	20491	50
world.htm	54863	554	40	10	24489	23870	50

Table 4: *Values for the top-set group of files. (Sizes in bytes, times in milliseconds.)*

file name	file size	number of URLs	scan time	stub creation time	hash table size	URLs size	time to serialize
index.htm	46960	360	27	10	22692	21400	60
default.htm	48419	380	33	10	24504	23870	50
01/index.htm	45504	369	95	10	22817	22444	60
02/index.htm	26753	200	16	20	14084	10789	40
03/index.htm	31834	279	22	10	18493	17033	50
04/index.htm	45247	360	26	10	21855	21656	50
05/index.htm	53778	411	30	10	25817	24490	60
06/index.htm	42476	362	25	10	22706	22081	70
01/default.htm	16843	150	20	10	8032	7934	10
02/default.htm	33473	173	24	10	8675	8090	30

Table 5: *Values for the branch-set group of files in the branch world/europe. (Sizes in bytes, times in milliseconds.)*

it takes to serialize a hash table with all the stubs corresponding to a single file.

However, in a normal browsing session, the user does not makes-replica of all the files. We expect the user to browse a few top-level pages and then pick one or more branches of the hierarchy. Some of these files will be replicated into the users local computer.

So, in order to obtain more realistic numbers, we performed the following. We picked 10 files from the top of the cnn.com hierarchy. These files are mostly entry points to the others with more specific contents. We call this set of files, the top-set. We also picked other 10 files representing a branch of the cnn.com hierarchy. We call this set of files, the branch-set.

In Tables 4 and 5, for each file in the top-set and in the branch-set, respectively, we present the times mentioned above along with the size of each file and the number of URLs enclosed.

These performance results are worst-case because they assume all the URLs enclosed in a file refer to a file in another site, which is not the usual case. However, they give us a good notion of the performance limits of the current implementation. In particular, we see that the most relevant performance costs are due to the scanning of a file and the serialization of the hash table. However, we believe that these values are acceptable taking into account the functionality of the system, i.e. it ensures that no dangling references and no memory leaks occur. In addition, when a user runs the NG browser and accesses any

web page without making a local replica of any file, there is absolutely no performance overhead due to DGC.

We can also conclude that the size on disk of the hash table containing all the stubs for a file is about half the size of the HTML file. This rather large size is mostly due to the size of the URLs which are responsible for about 90% of that size. The size of the file containing the stubs can certainly be reduced using regular compression techniques.

7 Related work

Much previous work in distributed garbage collection, such as SSP Chains [16] or Network Objects [4, 18], considers processes communicating by messages (without shared memory), using a hybrid of tracing and counting. Each process traces its internal pointers; references across process boundaries are counted as they are sent in messages.

Some object-oriented databases use a similar approach [2, 6, 21], i.e. a partition can be collected independently from the rest of the database. In particular, Thor is a research OODB [13] that stores data in a small number of servers. This data is cached at workstations for processing. A Thor server counts references contained in objects cached at a client. Thor defers counting references originating from some object x cached at a client, until x is modified at the server.

The work most directly related to this one is Skubiszewski and Porteix's GC-consistent cuts [17]. They consider asynchronous tracing of an object-oriented database, with no distribution or replication. The collector is allowed to trace an arbitrary database page at any time, subject to the following ordering rule. For every transaction accessing a page traced by the collector, if the transaction copies a pointer from one page to another, the collector either traces the source page before the write, or traces both the source and the destination page after the write. The authors prove that this is a sufficient condition for safety and liveness.

Most previous work on garbage collection in shared memory deals either with multiprocessors [3, 8] or with a small-scale DSM [9, 11]. These authors make strong coherence assumptions, and they ignore the fundamental issue of scalability.

Yu and Cox [22] describe a conservative collector for the TreadMarks DSM system. It uses partitioned GC on a process basis; it is strongly integrated with TreadMarks and all messages are scanned for possible contained pointers.

Previous work in DGC as IRC [14], SSP chains [16] and Larchant [10] served as the starting point of the DGC algorithm presented in this paper. Our new algorithm builds on these previous two algorithms in such a way that combines their advantages: no need for causal delivery support to be provided by the underlying communicating layer (from the first two), and capability to deal with replicated objects (from Larchant).

8 Conclusions and Future Work

In this paper we presented a new DGC algorithm for a WARM. The algorithm is general enough to be widely applicable given the minimal assumptions of the underlying model.

The fundamental aspects of the DGC algorithm are the following.

- It does not interfere with the protocol that maintains the replicas coherent among the participating processes. This means that the DGC does not require replicas to be coherent.
- It does not require causal delivery to be supported by the underlying communication layer. Given that supporting causal delivery in wide area networks is difficult and inefficient, this is a fundamental aspect to ensure the DGC algorithm scalability.
- It is safe in presence of replicated objects, i.e. it respects the union rule.

We presented our DGC algorithm as an evolution of two previous ones: a classical one designed for distributed systems based on function-shipping with no replication support, SSP chains, and Larchant which is targeted to distributed systems with replicated objects. However, it's important to note that any classical distributed garbage collection algorithm based on reference-counting can be used instead of SSP Chains (e.g. IRC). The only requirement would be its integration with the WARM in such a way that the safety rules are respected.

Concerning future research directions, we intend to address the fault-tolerance of the DGC algorithm. In other words, we are starting to study how the DGC algorithm should be designed so that it can remain safe, live and complete in spite of process crashes and permanent communication failures. We are also investigating how the DGC algorithm is affected if the WARM is accessed using transactions.

References

[1] Saleh E. Abdullahi, and Graem A. Ringwood. Garbage collecting the internet: A survey of distributed garbage collection. *ACM Computing Surveys*, 30(3):330–373, September 1998.

[2] L. Amsaleg, M. Franklin, and O. Gruber. Efficient Incremental Garbage Collection for Client-Server Object Database Systems. In *Proc. of the 21th VLDB Int. Conf.*, Zürich, Switzerland, September 1995.

[3] Andrew W. Appel. Simple Generational Garbage Collection and Fast Allocation. *Software Practice and Experience*, 19(2):171–183, February 1989.

[4] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. *Software Practice and Experience*, S4(25):87–130, December 1995.

[5] Xavier Blondel, Paulo Ferreira, and Marc Shapiro. Implementing garbage collection in the PerDiS system. In *Proc. of the Eigth International Workshop on Persistent Object Systems: Design, Implementation and Use (POS-8)*, 1998.

[6] Jonathan E. Cook, Alexander L. Wolf, and Benjamin G. Zorn. Partition selection policies in object database garbage collection. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, pages 371–382, Minneapolis MN (USA), May 1994. ACM SIGMOD.

[7] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-The-Fly Garbage Collection: An Exercise in Cooperation. *Communications of the ACM*, pages 966–975, Vol. 21, N. 11, November 1978.

[8] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proc. of the 20th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Lang.*, pages 113–123, Charleston SC (USA), January 1993.

[9] Paulo Ferreira and Marc Shapiro. Garbage collection and DSM consistency. In *Proc. of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 229–241, Monterey CA (USA), November 1994. ACM.

[10] Paulo Ferreira and Marc Shapiro. Modelling a distributed cached store for garbage collection: the algorithm and its correctness proof. In *ECOOP'98, Proc. of the Eight European Conf. on Object-Oriented Programming*, Brussels (Belgium), July 1998.

[11] T. Le Sergent and B. Berthomieu. Incremental multi-threaded garbage collection on virtually shared memory architectures. In *Proc. Int. Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 179–199, Saint-Malo (France), September 1992. Springer-Verlag.

[12] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[13] U. Maheshwari and B. Liskov. Fault-tolerant distributed garbage collection in a client-server, object database. In *Proceedings of the Parallel and Distributed Information Systems*, pages 239–248, Austin, Texas (USA), September 1994.

[14] José M. Piquer. Indirect reference-counting, a distributed garbage collection algorithm. In *PARLE'91—Parallel Architectures and Languages Europe*, volume 505 of *Lecture Notes in Computer Science*, pages 150–165, Eindhoven (the Netherlands), June 1991. Springer-Verlag.

[15] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), September 1995.

[16] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapport de Recherche 1799, Institut National de Recherche en Informatique et Automatique, Rocquencourt (France), November 1992.

[17] Marcin Skubiszewski and Patrick Valduriez. Concurrent Garbage Collection in O_2. In *Proceedings of the 23rd VLDB Conference*, Athens Greece, 1997.

[18] Sun Microsystems Inc. Java^{TM} Remote Method Invocation Specification, Revision 1.50, JDK 1.2. Documentation supplied with JDK 1.2 FCS, Oct. 1998.

[19] Luis Veiga and Paulo Ferreira. World wide news gathering automatic management. In *2nd International Conference Enterprise Information Systems*, Stafford, UK, July 2000.

[20] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, Saint-Malo (France), September 1992. Springer-Verlag.

[21] V. Yong, J. Naughton, and J. Yu. Storage reclamation and reorganization in client-server persistent object stores. In *Proc. Data Engineering Int. Conf.*, pages 120–133, Houston TX (USA), February 1994.

[22] Weimin Yu and Alan Cox. Conservative garbage collection on distributed shared memory systems. In *16th Int. Conf. on Distributed Computing Syst.*, pages 402–410, Hong Kong, May 1996. IEEE Computer Society.

Multi-Dispatch in the *Java Virtual Machine*: Design and Implementation

Christopher Dutchyn* Paul Lu* Duane Szafron* Steven Bromling* Wade Holst◊

* *Dept. of Computing Science*
University of Alberta
Edmonton, Alberta, Canada, T6G 2E8
{dutchyn,paullu,duane,bromling}@cs.ualberta.ca

◊ *Dept. of Computer Science*
The University of Western Ontario
London, Ontario, Canada, N6A 5B7
wade@csd.uwo.ca

Abstract

Mainstream object-oriented languages, such as C++ and Java[1], provide only a restricted form of polymorphic methods, namely uni-receiver dispatch. In common programming situations, developers must work around this limitation. We describe how to extend the Java Virtual Machine to support multi-dispatch and examine the complications that Java imposes on multi-dispatch in practice. Our technique avoids changes to the Java programming language itself, maintains source code and library compatibility, and isolates the performance penalty and semantic changes of multi-method dispatch to the program sections which use it. We have micro-benchmark and application-level performance results for a dynamic *Most Specific Applicable* (MSA) dispatcher, a framework-based *Single Receiver Projections* (SRP) dispatcher, and a tuned SRP dispatcher. Our general-purpose technique provides smaller dispatch latency than programmer-written double-dispatch code with equivalent functionality.

1 Introduction

Object-oriented (OO) languages provide powerful tools for expressing computations. One key abstraction is the concept of a *type hierarchy* which describes the relationships among types. Objects represent instances of these different types. Most existing object-oriented languages require each object variable to have a programmer-assigned *static type*. The compiler uses this information to recognize some coding errors. The *principle of substitutability* mandates that in any location where type T is expected, any sub-type of T is acceptable. But, substitutability allows that object variable to have a different (but related) *dynamic type* at runtime.

Another key facility found in OO languages is method selection based upon the types of the arguments. This method selection process is known as *dispatch*. It can occur at compile-time or at execution-time. In the former case, where only the static type information is available, we have *static dispatch* (method overloading). The latter case is known as *dynamic dispatch* (dynamic method overriding or virtual functions) and object-oriented languages leverage it to provide polymorphism — the execution of type-specific program code.

We can divide OO languages into two broad categories based upon how many arguments are considered during dispatch. *Uni-dispatch* languages select a method based upon the type of one distinguished argument; *multi-dispatch* languages consider more than one, and potentially all, of the arguments at dispatch time. For example, Smalltalk [14] is a uni-dispatch language. CLOS [23] and Cecil [6] are multi-dispatch languages. Other terms, like multiple dispatch, are used in the literature. However, the term multiple dispatch is confusing since it can mean either successive uni-dispatches or a single multi-dispatch. In fact, in this paper, we compare multi-dispatch to double dispatch, which uses two uni-dispatches.

C++ [24] and Java [15] are dynamic uni-dispatch languages. However, for both languages, the compiler considers the static types of all arguments when compiling method invocations. Therefore, we can regard these languages as supporting static multi-dispatch. Figure 1 depicts both dynamic uni-dispatch and static multi-dispatch in Java.

Uni-dispatch limits the method selection process to consider only a single argument, usually the receiver. This is a substantial limitation and standard programming idioms exist to overcome this restriction. As a motivation for multi-dispatch, we describe one programming idiom that demonstrates the need for multi-dispatch, describe

[1] Java is a trademark of Sun Microsystems, Inc.

```
class Point {
 int x, y;
 void draw(Canvas c) { // Point-specific code }
 void translate(int t)          {x+=t; y+=t;}
 void translate(int tX,int tY) {x+=tX; y+=tY;}
}
class ColorPoint extends Point {
 Color c;
 void draw(Canvas C) { // ColorPoint code }
}
// same static type, different dynamic types
Point Pp = new Point();
Point Pc = new ColorPoint();
// static multi-dispatch
Pp.translate(5); // one int version
Pp.translate(1,2); // two int version
// dynamic uni-dispatch
Pp.draw(aCanvas); // Point::draw()
Pc.draw(aCanvas); // ColorPoint::draw()
```

Figure 1: Dispatch Techniques in Java

how it can be replaced by multi-dispatch, list the advantages of using multi-dispatch to replace the idiomatic code, and measure the cost of using multi-dispatch with one of our current multi-dispatch algorithms.

1.1 Double Dispatch

Double dispatch occurs when a method explicitly checks an argument type and executes different code as a result of this check. Double dispatch is illustrated in Figure 2(a) (from Sun's AWT classes) where the `processEvent(AWTEvent)` method must process events in different ways, since event objects are instances of different classes. Since all of the events are placed in a queue whose static element type is `AWTEvent`, the compiler loses the more specific dynamic type information. When an element is removed from the queue for processing, its dynamic type must be explicitly checked to pick the appropriate action. This is an example of the well-known container problem [5].

Double dispatch suffers from a number of disadvantages. First, double dispatch has the overhead of invoking a second method. Second, the double-dispatch program is longer and more complex; this provides more opportunity for coding errors. Third, the double-dispatch program is more difficult to maintain since adding a new event type requires not only the code to handle the new event, but another cascaded `else if` statement.

The need for double dispatch develops naturally in several common situations. Consider binary operations [4], such as the `compareTo(Object)` method defined in interface `Comparable`. The programmer must ascertain the type of the `Object` argument before continuing to perform a type-specific comparison. Another common use for double dispatch is in drag-and-drop applications, where the result of a user action depends on both the data object dragged and on the target object. A generic drag-and-drop schema forces the programmer to test data types and re-dispatch to a more specific method. A third example is in event-driven programming. As we saw in Figure 2, applications are written using base classes such as `Component` and `Event`, but we need to take action based upon the specific types of both `Component` and `Event`. Indeed, the need for multi-dispatch is ubiquitous enough that two of the original design patterns, *Visitor* and *Strategy*, are work-arounds to supply multi-dispatch functionality within uni-dispatch languages.

Consider how the AWT example could be re-written if dynamic multi-dispatch was available in Java. An equivalent program, partially using multi-dispatch, would resemble Figure 2(b). For clarity, we have not completely converted the code to use multi-dispatch; we maintain the case statement and double dispatch to select among `MouseEvent` categories. A more complete factoring of `MouseEvent` into `MouseButtonEvent` and `MouseMotionEvent` would eliminate the remaining double dispatch, resulting in a *Full Multi-Dispatch* version of the code. The dynamic multi-dispatcher will select the correct method at runtime based upon the *dispatchable arguments* in addition to the *receiver argument* (the instance of `Component`). Individual component types can still override the methods that accept specific event types (e.g. `KeyEvent`, `FocusEvent`) and will do so without invoking the double-dispatch code.

The multi-dispatch version is shorter and clearer. However, it requires the Java Virtual Machine (JVM) [20] to directly dispatch an `Event` to the correct `processEvent(AWTEvent)` method. Our modified JVM provides this facility and correctly executes the multi-dispatch code discussed above. Furthermore, Table 1, a subset of Table 4, shows that multi-dispatch is substantially faster than interpreted double dispatch and even faster than JIT-ed double dispatch. Note that the numbers in Table 1 are based on single-threaded code.

Our experience with the Swing GUI classes [26] reinforces our belief that double dispatch in AWT is a significant factor in Swing applications. First, Swing does not operate without AWT; instead each `AWTEvent` is accepted by a Swing `JComponent`. Therefore, every mouse-click and key-press is double dispatched through AWT into Swing. Next, Swing type-checks the event and double dispatches again. Internally, Swing avoids further double dispatch by coding the `AWTEvent` type

```
package java.awt;
class Component {
 // double dispatch events to subComponent
 void processEvent(AWTEvent e) {
  if (e instanceof FocusEvent) {
   processFocusEvent((FocusEvent)e);
  } else if (e instanceof MouseEvent) {
   switch (e.getID()) {
   case MouseEvent.MOUSE_PRESSED:
    ...
   case MouseEvent.MOUSE_EXITED:
    processMouseEvent((MouseEvent)e);
    break;
   case MouseEvent.MOUSE_MOVED:
   case MouseEvent.MOUSE_DRAGGED:
    processMouseMotionEvent((MouseEvent)e);
    break;
   }
  } else if (e instanceof KeyEvent) {
   processKeyEvent((KeyEvent)e);
  } else if (e instanceof ComponentEvent) {
   processComponentEvent((ComponentEvent)e);
  } else if (e instanceof InputMethodEvent) {
   processInputMethodEvent((InputMethodEvent)e);
  }
  // other events ignored by Component
 }
 void processFocusEvent(FocusEvent e) {...}
 void processMouseEvent(MouseEvent e) {...}
 void processMouseMotionEvent(MouseEvent e) {...}
 void processKeyEvent(KeyEvent e) {...}
 void processComponentEvent(ComponentEvent e) {...}
 void processInputMethodEvent(InputMethodEvent e) {...}
}
```

(a) Double Dispatch in Java

```
package java.awt;
class Component {
 void processEvent(AWTEvent e) {...}

 void processEvent(MouseEvent e) {
  switch (e.getID()) {
  case MouseEvent.MOUSE_PRESSED:
   ...
  case MouseEvent.MOUSE_EXITED:
   processMouseEvent((MouseEvent)e);
   break;
  case MouseEvent.MOUSE_MOVED:
  case MouseEvent.MOUSE_DRAGGED:
   processMouseMotionEvent((MouseEvent)e);
   break;
  }
 }

 void processEvent(FocusEvent e) {...}
 void processMouseEvent(MouseEvent e) {...}
 void processMouseMotionEvent(MouseEvent e) {...}
 void processEvent(KeyEvent e) {...}
 void processEvent(ComponentEvent e) {...}
 void processEvent(InputMethodEvent e) {...}
}
```

(b) Equivalent Code in Multi-Dispatch Java

Figure 2: Double vs. Multi-Dispatch in Java

into the selector (e.g. `fireInternalEvent()`). Despite the limitations this imposes on the programmer, it is clear that double dispatch is still the standard technique in Swing as well.

Also, a multi-dispatch JVM could benefit other languages. For example, Standard ML, Scheme, and Eiffel have implementations which generate JVM-compatible binary files. Extending these languages to include multi-dispatch semantics becomes straightforward. Unlike techniques based on source code translation, our multi-dispatch JVM can be directly used by other languages.

The research contributions of this paper are:

1. The design and implementation of an extended Java Virtual Machine that supports arbitrary-arity multi-dispatch with the properties:
   (a) The Java syntax is not modified.
   (b) The Java compiler is not modified.
   (c) The programmer can select which classes should use multi-dispatch.
   (d) The performance and semantics of uni-dispatch methods are not affected.
   (e) The existing class libraries are not affected.
   (f) The existing reflection API is preserved.
2. The introduction of a dynamic version of Java's static multi-dispatch algorithm.
3. The first performance results for table-based multi-dispatch techniques in a mainstream language.

We begin by reviewing some important details about the uni-dispatch JVM. Next, we sketch our JVM modifications to enable multi-dispatch. Then, we present experimental results for implementations of our multi-dispatch techniques. This is followed by a discussion of several complex issues that a practical multi-dispatch Java must address and a description of some of the details of our implementation. Finally, we close with a description of future work and a review of related approaches to multi-dispatch.

2 Background

The Java Programming Language [15] is a static multi-dispatch, dynamic uni-dispatch, dynamic loading

Dispatch Type	Interpreter Time in μs	(σ)	Normalized	OpenJIT Time in μs	(σ)	Normalized
Double	0.91	(0.00)	1.00	0.48	(0.01)	1.00
Multi-	0.34	(0.00)	0.37	0.32	(0.01)	0.67
Full Multi-	0.32	(0.00)	0.35	0.32	(0.00)	0.67

Table 1: AWT Event Dispatch Comparison
(Call-site Dispatch Time in microseconds, Subset of Table 4)

object-oriented language. Our primary design goal is to extend the dynamic method selection to optionally and efficiently consider all arguments, without affecting the syntax of the language or any other semantics. Our secondary goals are to retain the dynamic and reflective properties of Java.

In order to meet these goals, we chose to modify the JVM [20] implementation, rather than modifying the programming language itself. Java programs are compiled by `javac` (or other compiler) into sequences of bytecodes — primitive operations of a simple stack-based computer. These bytecodes are interpreted by a JVM written for each hardware platform. We began with the *classic* VM (now known as the *Research Virtual Machine*[2]) written in C and distributed by Sun Microsystems, Inc. Other JVM implementations exist and many include *just-in-time* (JIT) compiler technology to enhance the interpretation speed at runtime by replacing the bytecodes with equivalent native machine instructions. At present, our modified JVM is compatible with the OpenJIT 1.1.15 [21] compiler.

Before we look at how to implement multi-dispatch in the virtual machine, we first need to understand the binary representation that the virtual machine executes, how method invocations are translated into the virtual machine code, and how the JVM actually dispatches the call-sites.

2.1 Java Classfile format

The JVM reads the bytecodes, along with some necessary symbolic information from a binary representation, known as a `.class` file. Each `.class` file contains a symbol table for one class, a description of its superclasses, and a series of method descriptions containing the actual bytecodes to interpret. We leverage the symbolic information, called the *constant pool*, to implement multi-dispatch.

Figure 3 shows the layout of the constant pool for the `ColorPoint` class shown in Figure 1.

[2]The Research Virtual machine was initially released as the *classic* reference VM. Sun later renamed it the *Exact* VM. With the advent of the *HotSpot* VM, the classic VM was renamed again, becoming the *Research* VM.

Conceptually, the constant pool consists of an array containing text strings and tagged references to text strings. In Figure 3, class `Point` is represented by a tag entry at location 1 that indicates that it is a CLASS tag and that we should look at constant pool location 2 for the name text. Then, the constant pool contains the text string "`Point`" at location 2. Therefore, a class symbol requires two constant pool entries. Method references are similar, except they require five constant pool entries.

1	CLASS	#2		Point
2	TEXT	"Point"		
3	CLASS	#4		ColorPoint
4	TEXT	"ColorPoint		
5	METHOD	#1	#6	Point::<init>:()V
6	NAME&TYPE	#7	#8	and for our initializer
7	TEXT	"<init>"		
8	TEXT	"()V"		
9	METHOD	#1	#10	Point::draw:(LCanvas;)V
10	NAME&TYPE	#11	#12	and for our method
11	TEXT	"draw"		
12	TEXT	"(LCanvas;)V"		
13	NAME&TYPE	#14	#15	used for our field
14	TEXT	"c"		
15	TEXT	"Color"		

Figure 3: A Simple Constant Pool

In our example, constant pool location 9 contains the tag declaring that it contains a METHOD. It references the CLASS tag at location 1, to define the static type of the class containing the method to be invoked. In this case, the class happens to be `Point` itself, but, more often, this is not the case. The METHOD entry also references the NAME-AND-TYPE entry at location 10. This NAME-AND-TYPE entry contains pointers to text entries at locations 11 and 12. The first location, 11, contains the method name, "`draw`". The second location, 12, contains an encoded signature "`(LCanvas;)V`" describing the number of arguments to the method, their types, and the return type from the method. In our example, we see one class argument with name "`Canvas`" and that the return type is `void`.

2.2 Static Multi-Dispatch in `Javac`

The Java compiler converts source code into a binary representation. When it encounters a method invocation, `javac` must emit a constant pool entry that describes the method to be invoked. It must provide an

exact description, so that, for instance, the two `translate(...)` methods in `Point` can be distinguished at runtime. Therefore, it must examine the types of the arguments at a call-site and select between them. This selection process, which considers the static types of all arguments, can be viewed as a static multi-dispatch.

The Java Language Specification, 2nd Edition (JLS) [15] provides an explicit algorithm for static multi-dispatch called *Most Specific Applicable* (MSA). At a call-site, the compiler begins with a list of all methods implemented and inherited by the (static) receiver type. Through a series of culling operations, the compiler reduces the set of methods down to a single most specific method. The first operation removes methods with the wrong name, methods that accept an incorrect number of arguments, and methods that are not accessible from the call-site. This latter group includes private methods called from another class and protected methods called from outside of the package.

Next, any methods which are not compatible with the static type of the arguments are also removed. This test relies upon testing *widening conversions*, where one type T_{sub} can be widened to another T_{super} if and only if T_{sub} is the same type as T_{super} or a subtype of T_{super}. For example, a `FocusEvent` can be widened to an `AWTEvent` because the latter is a super-type of the former[3]. The opposite is not valid: an `AWTEvent` cannot be widened to a `FocusEvent`; indeed a type-cast from `AWTEvent` to `FocusEvent` would need to be a type-checked *narrowing* conversion.

Finally, `javac` attempts to locate the single *most specific* method among the remaining subset of *statically applicable* methods. One method $\mathtt{M}(T_{1,1}, \ldots, T_{1,n})$ is considered more specific than $\mathtt{M}(T_{2,1}, \ldots, T_{2,n})$ if and only if each argument type $T_{1,i}$ can be widened to $T_{2,i}$ for each $(i = 1, \ldots, n)$, and for some j, $T_{2,j}$ cannot be widened to $T_{1,j}$. In effect, this means that any set of arguments acceptable to $\mathtt{M}(T_{2,1}, \ldots, T_{2,n})$ is also acceptable to $\mathtt{M}(T_{1,1}, \ldots, T_{1,n})$, but not vice versa.

Given the subset of applicable methods, `javac` selects one M_t as its tentatively most specific. It then checks each other candidate method M_c by testing whether its arguments can be widened to the corresponding argument in M_t. If this is successful, then M_c is at least as specific as M_t; the compiler adopts M_c as the new tentatively most specific method — the method M_t is culled from the candidate list. If the first test, whether M_c be widened to M_t, is unsuccessful, then the compiler checks the other direction: can M_t be widened to M_c. If so, then the compiler drops M_c from the candidate list.

Unfortunately, both tests can fail. To illustrate this, consider the first two methods in Figure 4. The first argument of the first method (`ColorPoint`) can be widened to the type of the first argument of the second method (`Point`). But the opposite is true for the second argument of each method. If we invoke `colorBox` with two `ColorPoint` arguments, both methods apply. If the third method was not present, we would have an *ambiguous method* error. The third method, taking two `ColorPoints`, removes the ambiguity because it is more specific than both of the other methods. It allows both of the others to be culled, giving a single most specific method.

```
colorBox(ColorPoint p1, Point p2) {...}
colorBox(Point p1, ColorPoint p2) {...}
        // conflict method removes ambiguity
colorBox(ColorPoint p1, ColorPoint p2) {...}
```

Figure 4: Ambiguous and Conflict Methods

Primitive types[4], when used as arguments, are tested at compilation time in the same way as other types. Primitive widening conversions are defined which effectively impose a standard type hierarchy on the primitive types. The compiler inserts widening casts as needed.

2.3 Dynamic Uni-Dispatch in the JVM

Now we turn our attention to dispatching polymorphic call-sites at runtime. Methods are stored in the `.class` file as sequences of virtual machine instructions. Within a stream of bytecodes, method invocations are represented by `invoke` bytecodes that occupy three bytes[5]. The first byte contains the opcode (`0xb6` for `invokevirtual`). The remaining two bytes form an index into the constant pool. The constant pool must contain a `METHOD` entry at the given index. This entry contains the static type of the receiver argument (as the `CLASS` linked entry), and the method name and signature (through the `NAME&TYPE` entry). Figure 5 shows the pseudo-bytecode[6] for invoking the method `Component.processEvent(AWTEvent)` twice.

From the opcode, `invokevirtual`, the JVM knows that the next two bytes contain the constant pool index of a `METHOD` descriptor. From that descriptor, the JVM can locate the method name and signature. The JVM parses the signature to discover that the method to be invoked requires a receiver argument and one other argument. Therefore, the JVM peeks into the operand

[3] The JLS separately recognizes identity conversions (a `FocusEvent` can be converted into a `FocusEvent`). `Javac` does not distinguish them, so we do the same for our exposition.

[4] Java provides non-object types `byte`, `char`, `short`, `int`, `long`, `float`, and `double`. These are called primitive types.

[5] The `invokeinterface` bytecodes occupy 5 bytes.

[6] Rather than show constant pool indices, we show their values directly.

```
Component aComponent = new SubComponent(...);
AWTEvent anEvent = new FocusEvent(...);
FocusEvent aFocusEvent = new FocusEvent(...);

aComponent.processEvent(anEvent);
aComponent.processEvent(aFocusEvent);
```

(a) Polymorphic Call–sites in Source.

```
...
apush     aComponent
apush     anEvent
invokevirtual    Component::processEvent:(LAWTEvent;)V

apush     aComponent
apush     aFocusEvent
invokevirtual    Component::processEvent:(LAWTEvent;)V
...
```

(b) Polymorphic Call–sites in Bytecodes.

Figure 5: Polymorphic Call-sites — two views

stack and locates the receiver argument. At this point, the JVM has the information it needs to begin searching for the method to invoke. The JVM has the name, the signature, and the receiver of the message.

The JVM Specification (section 5.4.3.3) provides a recursive algorithm for *resolving* a method reference and locating the correct method: Beginning with the methods defined for the precise receiver argument type, scan for an exact match for the name and signature. If one is not found, search the superclass[7] of the receiver argument, continuing up the superclass chain until `Object`, the root of the type hierarchy, is searched. If an exact match is not found, throw an `AbstractMethodError`. This look-up process applies to each of the `invoke` bytecodes.

This look-up process is a time-intensive operation. To reduce the overhead of method look-up, the resolved method is cached in the constant pool alongside the original method reference. The next time this method reference is applied by another `invoke` bytecode, the cached method is used directly.

Once a method is resolved, a method-specific *invoker* is executed to begin the interpretation of the new method. This invoker performs method-specific operations, such as acquiring a lock in the case of `synchronized` methods, constructing a JVM activation record in the case of bytecode methods, or preparing a machine-level activation record for `native` methods.

The Research JVM recognizes a special case in invoking methods: any private methods, final methods, or constructors can be handled in a *non-virtual* mode. Each of these situations do not require dynamic dispatch. But, multi-dispatch will need to handle these special cases.

[7] Java provides only single inheritance of program code.

3 Design

We now have sufficient information to describe the general design for extending the JVM to support multi-dispatch. In short, we mark classes which are to use multi-dispatch and replace their method invokers with one that selects a more specific method based on the actual arguments. Hence, existing uni-dispatch method invocations are unchanged in any way.

Marking the `.class` files without changing the language syntax is straightforward. We created an empty interface `MultiDispatchable` and any class which will provide multi-dispatch methods must implement that interface. The `.class` file retains that interface name and the virtual machine can easily check for this at class loading time. Our implementation does not change the syntax of the Java programming language or the binary `.class` file format in any way.

Our interface-based technique allows us to retain compatibility with existing programs, compilers, and libraries. Any class that implements our marker interface has different semantics for dispatch. But, the semantics of existing uni-dispatch programs and libraries are not changed since they do not implement the interface. The programmer retains complete control and responsibility for designating multi-dispatchable classes. This allows the developer to consciously target the multi-dispatch technique to known programming situations, such as double dispatch.

At dispatch time, our multi-invoker executes instead of the original JVM invoker. Our invoker locates a more-precise method based on the dynamic types of the invocation arguments and executes it in place of the original method.

The *non-virtual* mode invocations need to be handled specially. Constructors are never multi-dispatched. We found that constructor chaining within a class could cause infinite loops. Private and final multi-methods are still multi-dispatched.

We implemented two different dispatch algorithms. First, MSA implements a dynamic version of the Java Most Specific Applicable algorithm used by the `javac` compiler. Second, Single Receiver Projections (SRP) [17] is a high performance table-based technique developed at the University of Alberta. We examine both a framework-based SRP and a tuned SRP implementation. Section 6 provides implementation details, but we first present the results of our experiments.

4 Experimental Results

So far, we have used four different micro-benchmarks and a new implementation of Swing/AWT to test our multi-dispatcher.

The first micro-benchmark uses the `javac` compiler to recompile itself while running on the multi-dispatch VM. The `javac` compiler has not been modified, therefore the experiment demonstrates the backward compatibility of the modified VM for uni-dispatch applications. The measured overheads of uni-dispatch `javac` running on the multi-dispatch VM are minimal. The other three micro-benchmarks demonstrate multi-dispatch correctness, multi-dispatch performance as compared to double dispatch, and multi-dispatch performance as arity increases. All of the micro-benchmarks are single-threaded.

For our application-level tests, we modified Swing, the second-generation GUI library bundled with Java 2, to use multi-dispatch. As expected, Swing is a double-dispatch-intensive library. We also converted AWT because Swing depends heavily on AWT to dispatch the events into top-level Swing components.

All experiments were executed on a dedicated Intel-architecture PC equipped with two 550MHz Celeron processors, a 100MHz front-side bus, and 256 MB of memory. The operating system is Linux 2.2.16 with `glibc` version 2.1. The Sun Linux JDK 1.2.2 code was compiled using GNU C version 2.95.2, with optimization flags as supplied by Sun's makefiles[8]. The table-based multi-dispatch code [22] was compiled using GNU G++ version 2.95.2[9]. The Sun JDK only supports the `green` threading model, which is implemented using pthreads under Linux. We report average and standard deviations for 10 runs of each benchmark.

We tested three different virtual machines. First, we have *jdk*, the standard JDK 1.2.2 Linux runtime, running in interpreter mode. This JVM serves as a baseline for comparing the remaining four multi-dispatch systems. Second, we have a non-JIT multi-dispatch JVM with three different multi-dispatch techniques, *jdk-MSA*, and two implementations (*jdk-fSRP*, and *jdk-tSRP*) of the same algorithm. Third, we have customized OpenJIT 1.1.15 to be compatible with our multi-dispatch JVM.

For the first and second micro-benchmarks, (Tables 2 and 3) we report user+system time in seconds, along with normalized values against the *jdk* runtime. For the third and fourth experiments (Table 4 and Figure 7), we describe individual dispatch times in microseconds, ignoring other costs. In the final benchmark, Swing, we report execution times for a synthetic application that creates a number of components and inserts 200,000 events into the event queue.

[8] Typical flags are `-O2`

[9] with options `-ansi -fno-implicit-templates -fkeep-inline-functions -O2`.

4.1 `Javac` — Compatibility Test

The first experiment requires the runtime to load and execute the `javac` compiler to translate the entire `sun.tools` hierarchy of Java source files into `.class` files. This hierarchy includes 234 source files encompassing 49,798 lines of code (excluding comments). Each compilation was verified by comparing the error messages[10] and by checksumming the generated binaries. Each virtual machine passed the test; the timing results are shown in Table 2. These times come from the Unix `time` user command and are averages, with standard deviation, of 10 runs.

JVM	Time in sec.	(σ)	Norm.
jdk	65.41 + 0.25	(0.39)	1.00
jdk-MSA	67.38 + 0.31	(0.14)	1.03
jdk-fSRP	68.22 + 0.45	(0.25)	1.05
jdk-tSRP	67.13 + 0.51	(0.35)	1.03

Table 2: Compatibility Testing and Performance
(User+System Time to Recompile `sun.tools`, in seconds)

The negligible differences between the uni-dispatch and multi-dispatch execution times demonstrate that the overhead of running uni-dispatch code on a multi-dispatch VM is essentially zero. Note that in our implementation, table-based JVMs do not construct a dispatch table until the first multi-dispatchable method is inserted.

4.2 Simple Multi-Dispatch

In this micro-benchmark, we show that multi-dispatch is correct and measure its overhead. The testing code is short and is shown in Figure 6. Note that class `MDJDriver` implements the marker interface `MultiDispatchable`. The compiler uses static multi-dispatch to code all four calls to `MDJDriver.m(X,X)` to execute the method for two arguments of type `A`, because that is the static type of both `anA` and `aB`. Multi-dispatch actually selects among the four methods based upon the dynamic types of the arguments. Therefore, correct output consists of 100,000 repetitions of four consecutive lines: `AA`, `AB`, `BA`, and `BB`. For timing purposes, all output was redirected to `/dev/null` to reduce the impact of input/output. Our results are summarized in Table 3.

The table-based techniques, jdk-fSRP and jdk-tSRP, suffer from a substantial startup time, whereas `jdk-MSA`

[10] There is one warning noting that 8 files used deprecated APIs.

```
class A { }
class B extends A { }
class MDJDriver implements MultiDispatchable {
 String m(A a1, A a2) { return "AA"; }
 String m(A a1, B b2) { return "AB"; }
 String m(B b1, A a2) { return "BA"; }
 String m(B b1, B b2) { return "BB"; }
 static public void main(String args[]) {
  final int LOOPSIZE = 100000;
  A anA = new A();
  A aB = new B();
  MDJDriver d = new MDJDriver();
  for( int i=0; i<LOOPSIZE; i++) {
   System.out.println(d.m(anA, anA));
   System.out.println(d.m(anA, aB));
   System.out.println(d.m(aB, anA));
   System.out.println(d.m(aB, aB));
  }
 }
}
```

Figure 6: Simple Multi-Dispatch Testing Code

primarily uses existing data structures found in the JVM interpreter and lazily computes any additional values. This reduces the cost of program startup.

JVM	Time in sec.	(σ)	Norm.	Correct
jdk	26.40 + 0.68	(0.07)	1.00	No
jdk-MSA	28.88 + 0.83	(0.22)	1.10	Yes
jdk-fSRP	31.53 + 0.91	(0.11)	1.20	Yes
jdk-tSRP	29.48 + 0.84	(0.17)	1.12	Yes

Table 3: Simple Multi-Dispatch
(User+System Execution Time in seconds)

4.3 Double Dispatch of Events

Our third experiment involves computing the performance differences between double dispatch and the two multi-dispatch implementations of the example given in Figure 2. We constructed a synthetic type hierarchy of `AWTEvent` classes, to match those in Figure 2. The discussion of Swing follows in Section 4.5. We also constructed three different component types:

Double Dispatch (DD) implements double dispatch via type-cases and programmer-coded type numbering as shown in Figure 2(a).[11]

Multi-Dispatch (MD) implements multi-dispatch as shown in Figure 2(b), where the type-cases from `DD` have been replaced with multi-dispatch.

[11] Type-cases are not the most effective double-dispatch technique, but this code matches Sun's AWT implementation. For a comparison with other double-dispatch techniques, see [8, 13].

Full Multi-Dispatch (FMD) eliminates the type-cases and the programmer-coded type-numbering from `DD`. It divides `MouseEvent` into two different classes and eliminates the `switch` statement.

To avoid inlining effects, we added code for updating an instance variable to the body of each `processEvent(AWTEvent)`. This experiment consists of dispatching a total of one million events through `processEvent(AWTEvent)`. Each event type appears equally often, as we iterate over an array containing equal numbers of each event. We compute the loop overhead, subtract the overhead amount, and then divide the remaining time by the number of events dispatched. The timing results are shown in Table 4.

Also, we give an additional timing value for our custom SRP implementation, where we disabled mutual exclusion in the dispatcher. Currently our implementation uses a costly monitor to ensure that no other thread is updating the dispatch tables during a multi-dispatch. High-performance concurrent-read exclusive-write protocols can eliminate this overhead; the nolock value represents this highest-performance case.

As DD does not declare itself multi-dispatchable, the similarity of the results in column 2 of Table 4 again shows that our multi-dispatchable virtual machines do not significantly penalize uni-dispatch code. Further, we see that the cost of interpreting numerous expensive JVM bytecodes, such as `instanceof`, followed by another `invokevirtual` (which is DD's strategy), is more costly than our multi-dispatch techniques. The full multi-dispatch implementation (FMD) is faster than the partial multi-dispatch (MD). This is reasonable because MD ends up double-dispatching two of every six events.

Again, we see that the framework-based SRP technique suffers from considerable initial overhead. We hypothesize that it is a result of the object-oriented nature of our implementation of the table-based techniques. In each dispatch, several C++ objects are created and destroyed on the heap. Our tuned SRP implementation, jdk-tSRP, removes this overhead and provides faster dispatch performance than programmer-coded double dispatch.

OpenJIT compilation gains only minor improvements for the multi-dispatch system. This matches our expectations since OpenJIT calls the same `selectMultiMethod()` routine that the interpreter uses, there is only a slight benefit from avoiding some interpreter frame manipulations.

4.4 Arity Effects

Our final micro-benchmark explores the time penalties as the number of dispatchable arguments and applicable

Dispatch	Interpreter						OpenJIT					
	DD		MD		FMD		DD		MD		FMD	
JVM	Time	(σ)	Time	(σ)	Time	(σ)	Time	(σ)	Time	(σ)	Time	(σ)
jdk	0.91	(0.00)	—		—		0.48	(0.00)	—		—	
jdk-MSA	0.95	(0.00)	2.63	(0.01)	2.49	(0.02)	0.95	(0.00)	2.55	(0.04)	2.43	(0.03)
jdk-fSRP	0.96	(0.01)	3.12	(0.08)	2.52	(0.05)	0.96	(0.01)	2.90	(0.05)	2.47	(0.05)
jdk-tSRP	0.94	(0.00)	0.75	(0.03)	0.72	(0.02)	0.95	(0.00)	0.74	(0.02)	0.71	(0.01)
nolock	0.95	(0.00)	0.34	(0.00)	0.32	(0.00)	0.95	(0.00)	0.32	(0.01)	0.32	(0.00)

Table 4: Event Dispatch Comparison
(Call-site Dispatch Times in microseconds)

methods grow. To do this, we built a simple hierarchy of five classes (one root class A, with three subclasses B, C, and D, and finally class E as a subclass of C) and constructed methods of different arities against that hierarchy. We defined the following methods:

- classes A, B, C, D, and E contain unary methods `R.m()` (where *R* represents the receiver argument class).
- classes A, B, C, D, and E also implement five binary methods, `R.m(X)` where *X* can be any of A, B, C, D, or E.
- classes A, B, C, D, and E implement 25 ternary methods, `R.m(X, Y)` where *X* and *Y* can be any of A, B, C, D, or E.
- classes A, B, C, D, and E implement 125 quaternary methods, `R.m(X, Y, Z)` where *X*, *Y*, and *Z* can be any of A, B, C, D, or E.

MSA looks at one fewer dispatchable arguments than the table-based techniques because the receiver argument has already been dispatched by the JVM. For instance, given a unary method, MSA makes no widening conversions for dispatchable arguments. A binary method requires MSA to check only one widening conversion. The table-based techniques dispatch on all arguments and gain no benefit from the dispatch done by the JVM.

We invoke one million methods for each arity. This means that each of the unary methods is executed 200,000 times. However each of the quaternary methods is executed only 1,600 times. After computing the loop overhead via an empty loop, we determine the elapsed time to millisecond accuracy and determine the time taken for each dispatch. Our results are shown in Figure 7.

We can evaluate the arity effects in the uni-dispatch case by coding a third level of double dispatch. Already the overhead of constructing a third activation record exceeds the dispatch time of our tuned SRP implementation. Also, our SRP implementations suffer only linear growth in time-penalties as arity increases, whereas MSA suffers quadratic effects.

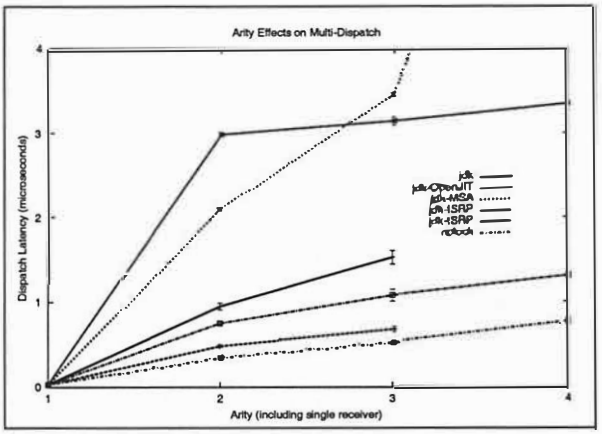


Figure 7: Impact of Arity on Dispatch Latency

4.5 Swing and AWT

Our final test is to apply multi-dispatch to AWT and Swing applications. To do this, we needed to rewrite AWT and Swing to take advantage of multi-dispatch.

We modified 11% (92 out of 846) of the classes in the AWT and Swing hierarchies. We eliminated 171 decision points, but needed to insert 123 new methods to replace existing double-dispatch code sections. Within the modified classes, we removed 5% of the conditionals and reduced the average number of choice points per method from 3.8 to 2.0 per method. This reduction illustrates the value of multi-dispatch in reducing code complexity.

In all, 57 classes were added, all of them new event types to replace those previously recognized only by a special type id (as in the AWT examples described previously). Our multi-dispatch libraries are a drop-in replacement that executes a total of 7.7% fewer method invocations and gives virtually identical performance with applications such as `SwingSet`. In our sample application, we found that the number of multi-dispatches executed almost exactly equaled the total reduction in method in-

Stage	Uni-Swing Methods	Multi-Swing Uni-Methods	Multi-methods
warm-up	901,938	901,795	160 (0.02%)
event loop	32,543,684	27,807,327	2,350,172 (7.7%)

Table 5: Swing Application Method Invocations

vocations. This suggests that every multi-dispatch replaced a double dispatch in the original Swing and AWT libraries.

We verified the operation of the entire unmodified `SwingSet` application with our replacement libraries. Finally to measure performance, we timed a simple Swing application that handles 200,000 `AWTEvents` of different types. The timing results are given in Table 6.

Dispatch JVM	Uni-Swing Time	(σ)	Multi-Swing Time	(σ)
jdk	28.03	(0.35)	—	
jdk-MSA	28.69	(0.31)	70.09	(0.15)
jdk-tSRP	29.33	(0.42)	28.30	(0.36)

Table 6: Swing Application Execution Time
(Event loop times in seconds)

The Swing and AWT conversion also demonstrates the robustness of our approach. We needed to support multi-dispatch on instance and static methods. Nolock values are not given because Swing breaks our simplification that dispatch tables are not updated concurrently, and jdk-fSRP values are not given because the framework-based system does not support static methods. Swing and AWT expect to dispatch differently on `Object` and array types. In modifying the libraries, we found numerous opportunities to apply multi-dispatch to private, protected, and super method invocations. In addition, several multi-methods required the JVM to accept covariant return types from multi-methods. All of these features are required for a mainstream programming language.

5 Multi-Dispatch Issues

Besides performance and correctness, multi-dispatch must contend with a number of serious difficulties which the `javac` compiler cannot recognize. They are: ambiguous method invocations caused by inheritance conflicts, incompatible return type changes, masking of methods by primitive widening operations, and null arguments. Each of these is illustrated in Figure 8. We have developed a tool called `MDLint` that can identify these problems and warn the programmer.

The first difficulty is that multi-dispatch, even in a single-inheritance language, can suffer from ambiguous methods. The two examples using the `m1` methods illustrate this. For the first method invocation, the compiler knows that `A.m1(B)` and `B.m1(A)` are candidates. Neither one is more specific than the other, so the compiler aborts with an error. We can fix that by statically typing the receiver argument to `A`, but multi-dispatch sees exactly the same conflict at runtime. Our `MDLint` program warns about the problem. If the programmer disregards the warning, our JVM detects the error and throws an `AmbiguousMethodException`.

Throwing a runtime exception may seem neither elegant nor acceptable, but one of the key attributes of the JVM is to maintain security. A malicious programmer can separately compile each class so that errors are not evident until execution. The JVM must protect itself from these possibilities, and throwing an exception is the only option. As we noted, our `MDLint` tool can recognize and report potential ambiguities, exception inconsistencies and return-type conflicts at compile time.

The second difficulty centers around the fact that `javac` considers methods with different argument types as distinct. This means that they can have different return types. Multi-dispatch forges additional connections between classes based on the additional dispatchable arguments. This means that methods which `javac` considered distinct are now overriding each other. In the example, we see that the two `m2(...)` methods override each other for multi-dispatch. Our multi-dispatch implementations throw an `IllegalReturnTypeChange` exception, unless the more specific method returns a subtype of the original returned value.

Another ramification of the fact that uni-dispatch Java considers different argument combinations as distinct methods is that `javac` does not ensure that the `throws` clauses are compatible. As with any overriding method, we would want a more specific multi-method to covariantly-specialize the set of exceptions. Our type-checker validates this, but, in compliance with the VM specification, our virtual machine neither checks nor reports this inconsistency.

The third difficulty involves the use of literal null as an argument. If null is typed, as in the first invocation of `m3()`, then `javac` performs static multi-dispatch with that type. This restricts the set of applicable methods `javac` will consider. In our example, an ordinary JVM can avoid loading class `C`. The multi-dispatch JVM recognizes that `m3(C)` might apply (since `a` is dynamically

```
class A {
 void m1(B b1) {...}
 void m4(int i) {...}}
class B extends A {
 void m1(A a1) {...}
 void m4(byte b) {...}}
class C extends B {...}
class MDJIssues {
 int m2(A a1, A a2) {...}
 String m2(B b1, B b2) {...}
 void m3(A a1) {...}
 void m3(B b1) {...}
 void m3(C c1) {...}
 public static void main(String args[]) {
  A Ab = new B(); // static:  A, dynamic:  B
  B Bb = new B(); // static:  B, dynamic:  B
  // multi-dispatch difficulties
  Bb.m1(Bb); // javac:  ambiguous method
  Ab.m1(Bb); // javac:  OK, MDJ: ambiguous
  // incompatible return type change
  int i = m2(Bb, Bb); // javac:  bad return type
  int j = m2(Ab, Ab); // javac:  OK, MDJ: exception
  // null arguments are more consistent
  A a = null;
  m3(a); // regular Java:  executes m3(A)
         // MDJ: loads C, executes m3(C)
  m3(null); // both execute m3(C)
  // stronger referential integrity
  m3(Ab); // regular Java:  executes m3(A)
          // MDJ: executes m3(B)
  m3(new B()); //both execute m3(B)
  // primitive widening hides correct method
  byte b = 7;
  Ab.m4(b); // javac:  widens, calls A.m4(int)
       // MDJ: ignores B.m4(byte), calls A.m4(int)
  Ab.m4(int(b)); // programmer widening
 }
}
```

Figure 8: Examples of Multi-Dispatch Issues

of null type and null is subtype of class C). Therefore, multi-dispatch Java loads class C in order to determine its place in the type hierarchy, and decides that `m3(C)` is the most-specific method. Literal nulls, as shown in the second invocation of `m3()`, illustrate the inconsistency of standard Java; it now agrees with the multi-dispatch JVM that `m3(C)` should be invoked. The ordinary JVM can still avoid loading class C, because `javac` has already static multi-dispatched to `m3(C)`[12]. Presumably, the argument is used in `m3(C)`, so the ordinary JVM will end up loading class C, just like the multi-dispatch JVM.

The null argument problem is an example of a more general referential transparency problem in Java. Inconsistent invocations can occur when expressions are substituted in place of variables. This is because `javac` might apply more precise type information from the substituted expression. As an example, compare the execution of the third and fourth invocations of `m3(...)`. By replacing `Ab` with its value, we have altered the execution of a program.

The last difficulty is more complex and, at this time, unsolved. The compiler selects a method based upon widening operations and may change the type of primitive arguments. In the example, the compiler inserts instructions to convert `b` from a `byte` to an `int`. At runtime, we have lost all traces that `b` was originally specified as a `byte`. Indeed, the programmer might have wanted to force that exact conversion; the bytecodes would be identical to compiler-generated conversions.

6 Implementation

In this section, we describe how the JVM is extended to support dynamic multi-dispatch. We begin by examining how to indicate to the JVM which classes are multi-dispatchable. We then examine where multi-dispatch must occur and, finally, we review three different multi-dispatch implementations.

6.1 Marking Multi-Dispatch Classes

We tell the JVM that multi-dispatch is required on a class-by-class basis by implementing the empty interface `MultiDispatchable` in each class that is multi-dispatchable. The Java programming language has already leveraged this idea for marking class capabilities with the `Cloneable` interface. We use the `MultiDispatchable` interface to denote that any method sent to a multi-dispatch receiver should be handled by the multi-dispatcher. For efficiency, we add a flag to the internal class representation to indicate that a class is multi-dispatchable, rather than searching its list of interfaces at each method invocation. The value of this flag is set once, at class load time.

Our selection of `MultiDispatchable` as the marker requires us to recognize multi-dispatch on a class-by-class basis, not on a method-by-method or argument-by-argument basis. That is, every method invocation where the uni-dispatch receiver is a member of a multi-dispatchable class goes through our multi-dispatcher. Furthermore, because interfaces are inherited, this approach requires any subclass of a multi-dispatchable class to also be multi-dispatchable. Most importantly, any method invocation where the receiver argument is not marked for multi-dispatch continues unchanged through the uni-dispatcher. The benefit of this is that the syntax of Java programs is unchanged, and the performance and semantics of uni-dispatch remains intact.

The techniques used to *mark* code as multi-dispatchable and to *implement* multi-dispatch method invocations are independent. `MultiDispatchable` marks entire

[12] There is a subtlety here because `javac` selects the most-specific method from the method dictionary of the static type of the receiver. Therefore, dynamic uni-dispatch still may not select the most-specific method of the receiver's dynamic class.

classes without language extensions, but our JVM actually supports multi-dispatch on a method-by-method basis. An alternate tagging mechanism, that marked individual methods as multi-dispatchable, may be possible if we permitted language extensions.

6.2 Adding Multi-Dispatch

As part of the uni-dispatch of an `invoke` bytecode, the JVM finds a method pointer from the array of methods in the receiver argument class. At this point, the interpreter loop is about to build a new frame to execute the found method. The interpreter loop (and classic VM JIT compilers) proceed to call a special function, called the `invoker` that handles the details of building the new frame and starting the new method. The Research JVM uses different invokers for native, bytecode, synchronized, JIT-compiled, and other method types. Similar to the OpenJIT system [21], we replace this invoker function with a custom *multi-invoker* that computes the correct multi-dispatch method. Once the more precise method is known, we simply invoke it directly.

The multi-invoker is installed at class-load time. The interpreter loop and invoker for uni-dispatch are unchanged. This supports our claim that uni-dispatch programs and libraries suffer no execution time penalties.

OpenJIT is supported in exactly the same way. Every method contains a `compiledCode` function pointer onto which OpenJIT installs its compiled method body. Once the compilation is complete, OpenJIT saves the compiled method body of any multi-method to a new field `oldCompiledCode` and installs a pointer to a routine `DispatchMulti()`. This replacement invoker simply calls the same method specializer `selectMultiMethod()` that the interpreter uses. If the more precise method-body is already compiled, then OpenJIT jumps into the `oldCompiledCode`, executing the more specific compiled method. Alternately, if the more precise method is not already JIT-ed, then `DispatchMulti()` sets it to be compiled and invokes the interpreter on the bytecode version.

Unfortunately, we must disable much of the inlining facility of OpenJIT when using multi-dispatch. The uni-dispatch OpenJIT compiler can inline `private`, `static`, and `final` methods because they can never change. With multi-dispatch, this is no longer true — at a given call-site, the selected multi-method may change depending on the arguments to the current invocation. The JIT compiler and VM must work together to ensure that every method invocation is checked for multi-dispatch and correctly specialized.

The core component of our system is the `selectMultiMethod()` routine, which locates a more-specific method applicable to a set of arguments. We have experimented with three different multi-dispatch techniques; they are examined in the following sections. For each technique, we also describe our solution for the implementation issues described in section 5.

6.3 Reference Implementation: *MSA*

Our reference implementation is an extension of the Most Specific Applicable algorithm described in section 15.11 of *The Java Language Specification* and in section 2.2 of this paper. In particular, we re-examine the steps described in section 2.2 in light of the dynamic argument types being used.

When the multi-invoker is called, it has access to the `methodblock` that has already been found by the uni-dispatch resolution mechanism. We also have the top of the operand stack, so we can peek at each of the arguments. Last, we have the actual receiver, which can provide the list of methods (including inherited ones) that it implements.

Every method is represented by a `methodblock` containing many useful pieces of information. First, it holds the name of the method. Second, it contains a handle to the class that contains this method[13]. Third, it contains the signature which we can parse to get the arity and types of the dispatchable arguments. For performance, we parse the signature only once. We add two fields to the `methodblock`: `int arity` to cache the arity and `ClassClass **argClass` to hold the class handles for the dispatchable arguments.

With these three pieces of information, we implement a dynamic version of the MSA algorithm directly. Wherever the original algorithm would use the static type of an argument, we apply the known dynamic type instead. In step 2(b) from section 2.2, the compiler would compare the static type of each argument with the corresponding declared type for the candidate method. In the dynamic case, we have the arguments on the stack, so we can find their dynamic types. We compare each argument's dynamic type against the declared type of the corresponding argument of the method. We discard any method that is not applicable due to access rights (`private` methods) or whose declared types do not match the arguments on the stack. The remaining methods are *dynamically applicable*.

The issue of null-valued arguments becomes significant at this point. JLS chapter 4 recognizes the need for a *null type* to represent (untyped) null values. It further

[13] Recall that methods might be inherited; this class handle is the original implementing class.

declares in section 4.1 that the null type can be coerced to any non-primitive type. Also, section 5.1.4 allows null types to be widened to any object, array or interface type. Statically, this means that an (untyped) null argument can be widened to any class. In the dynamic case, we want to do the same. Therefore, whenever we encounter a null argument we accept the conversion of that null to a method argument of type class, array, or interface.

Unfortunately, if we have a null argument, we may retain a method which accepts arguments of classes that are not yet loaded. We need to force these classes to be loaded to ensure that the next step operates correctly.

Given the list of applicable methods, step 2(d) finds the unique most specific method. Again the operation is identical to the process that the `javac` compiler follows. One applicable method is tentatively selected as the most specific. Each other applicable method is tested by comparing argument by argument (including the receiver argument) against the tentatively most specific. At each step, we discard any methods that are less specific. We continue this process until only one candidate method remains, or two or more equally specific methods remain. In the latter case, we have an ambiguous method invocation and we throw an `AmbiguousMethodException` to advertise this fact.

Next, we verify that the return type for our more specific method is compatible with the compiler-selected method. This check relaxes JLS 8.4.6.3, where we must reject any invocation that has a different return type, yet ensures type-safety. If the return type is different, we throw an `IllegalReturnTypeChange` exception at runtime.

6.4 Table-based Dispatch

Our SRP framework-based techniques is taken from the Dispatch Table Framework (DTF) [22]. This is a toolkit of many different uni-dispatch and multi-dispatch techniques. In order to call the DTF to dispatch a call-site, we need to inform the DTF of the various classes and methods present in our Java program. Our interface consists of a number of straight-forward routines to perform this registration.

The JVM maintains in-memory structures for each loaded `.class` file. We have extended that `ClassClass` structure to contain a `DTF_Type` field. It contains a pointer to the C++ object generated by the DTF. Once a class is dynamically loaded by the JVM, we check to see if we must register it with the dispatcher. If the dispatcher has already been instantiated, we register the class via `javaAddClass(...)` and store away the returned `DTF_Type` pointer.

If a dispatcher has not been instantiated, and the just-loaded class is uni-dispatch only, we defer the registration in order to reduce the overhead to uni-dispatch programs. If the just-loaded class is marked for multi-dispatch and the dispatcher has not been instantiated, the process is more complex. First, we instantiate a new dispatcher. Then, we register each class that has already been loaded, ensuring that its superclasses and superinterfaces are registered first.

Finally, as the last part of registering a class with the dispatcher, we need to see whether any methods from other classes were held in abeyance until this class was loaded. This can occur if the methods from other classes expect dispatchable arguments of the class we are just now loading. As we shall see below, we deferred registering these methods until the class was loaded.

Java's facility for dynamically reloading classes forces us to ensure that two classes with the same name are assigned different `DTF_Types`. Java ensures that two classes with the same name are treated as distinct by insisting that each one is loaded by a different classloader [19]. We apply the same technique by supplying the DTF framework with a name consisting of the classloader name, followed by "`::`" and followed by the class name. They system classloader is given the empty name "".

For a class marked for multi-dispatch, we need to register its methods along with their types, via `javaAddMethod(...)`. If this class implements `MultiDispatchable` directly, then we register all of its methods, including inherited ones. Alternately, if `MultiDispatchable` is an inherited interface for this class, then we know that its superclass has already registered its methods. Therefore, we do not need to register them; we only need to register the methods that we directly implement.

This method registration process is complicated by our desire to load classes lazily. If a method accepts an argument with a class not yet seen by the JVM, we know that we could never dispatch to it until that class is loaded[14]. We set that method aside for future registration.

If all of the argument types for the method are already registered with the DTF, then we proceed to register the method. We provide a `methodblock` pointer that we want the framework to return if this method is the dispatched target. We bundle up the `DTF_Type` values found in the `ClassClass` structures for each argument class (including the receiver argument) and pass them to the framework. The framework returns a

[14] As mentioned above, our DTF-based systems do not permit null as a dispatchable argument. Therefore, this guarantee holds.

DTF_Behavior pointer that we store in the methodblock.

Dispatch becomes a very simple operation. We build an array of the DTF_Type pointers from the arguments on the Java stack. If we encounter a null argument, we throw a NullPointerException. The DTF_Type array, along with the DTF_Behavior pointer from the compiler-selected method allow the framework to locate the methodblock pointer that we had previously registered.

We expect that the returned methodblock pointer is the method for multi-dispatch. We validate it against the compiler-selected method. If the return type has changed, we abort the dispatch and throw an IllegalReturnTypeChange exception. Otherwise, we call the found method's original invoker and return its value as the result of the interpreter's call to a method invoker.

Single Receiver Projections Single Receiver Projections (SRP) [16] is a technique that considers a multi-dispatch as a request for the joint most specific method available on each argument. For a given argument position and type, an ordered (most-specific to least-specific) vector of potential methods is maintained. The vectors for all the argument positions are intersected to provide an ordered vector of all applicable methods. Because of the ordering, this vector can be quickly searched for the most applicable method.

SRP uses a uni-dispatch technique to maintain the vector of potential methods for each individual argument. These vectors are typically compressed to conserve space. Many different compression techniques are known: row displacement, selector coloring [2], and compressed selector table indexing [25]. Our implementation uses selector coloring, because timing experiments [17] indicates that technique provides the fastest dispatch times.

7 Future Work

Our MSA and tuned SRP dispatchers are the most complete. They support null as a dispatchable argument, multi-dispatch on other invoke bytecodes[15], widening of primitive dispatchable arguments, and multi-threaded dispatch. Our table-framework-based dispatchers do not currently support all of these facilities. Adding them would provide additional flexibility and allow them to fully support the Java programming language semantics. In particular, we have a two-table design that will allow one thread to dispatch through an existing table, while we register additional methods and/or classes to a new one.

Our custom SRP code implements multi-dispatch as a critical section, protected by a mutual-exclusion lock. We have devised, but not as yet implemented, a technique which would eliminate the lock overhead (approximately 0.38 μs for every multi-dispatch) and allow concurrent multi-dispatch. The trade-off is that every thread would need to halt while the multi-dispatch tables are being updated.

The OpenJIT support for multi-dispatch is still primitive; in particular, we eliminate all inlining actions. This is a conservative approach and one can identify situations where inlining in multi-dispatch Java would provide correct results. Identifying these opportunities will yield higher overall performance.

Other multi-dispatch techniques exist, including compressed n-dimensional tables [1, 12], look-up automata [9, 10], and efficient multiple and predicate dispatch [7]. A comprehensive exploration of these techniques using Java is incomplete at this time.

Another significant improvement for multi-dispatch is to incorporate our code testing tool into the javac compiler. At this time, MDLint exists as a separate executable which will recognize and warn the programmer about common ambiguities and difficulties. It analyzes a complete application and identifies the code sections where the programmer could invoke an ambiguous method, or have a conflicting return type.

Our reference implementation, MSA, supports multi-dispatch on all method types (instance, static, interface, private, etc.), except constructors. Because the same bytecode is used to invoke a constructor in the superclass and a constructor with different arguments, we cannot distinguish the two possibilities. This issue is a specific instance of the need to apply a super to an argument other than the receiver. Fortunately, in our experience, this requirement does not arise in common programming practice (except for constructors).

Our tuned SRP implementation allows our dispatch tables to identify only those types that are multi-dispatched. This *lazy type numbering* is reversible, allowing the tables to shrink as classes are unloaded. In turn, multi-methods can revert to lower arity multi-dispatch (or even uni-dispatch). We see great promise in this technique for long-lived Java server applications.

The DTF framework contains another dispatcher, *Multiple Row Displacement* [22] (MRD) that operates 15% faster than SRP. Therefore, we expect that dispatch could be enhanced to provide even lower latency by applying this technique. Unfortunately, MRD currently does not

[15] Signaled by implementing the empty interfaces StaticMultiDispatchable and SpecialMultiDispatchable.

support incremental dispatch table updates in the same way that SRP does. In a dynamic environment such as Java, incremental updating of dispatch tables is desirable. Enhancing MRD to support incremental updates is another research priority.

Last, our marker interface `MultiDispatchable` denotes that each method in a given class is to be multi-dispatched. Our JVM relies on this tag only to inform it about which methods are eligible for multi-dispatch. Therefore, without changing our multi-dispatch implementation, alternate Java syntax would allow us to selectively mark individual methods (and their overriding multi-methods) as multi-dispatchable, rather than entire classes. We would like to explore the space of conservative language extensions to expose this feature.

8 Related Work

Others have attempted to add multi-dispatch to Java through language preprocessors. Boyland and Castagna [3] provide an additional keyword *parasite* to mark methods which should have multi-dispatch properties. They effectively translate these methods into equivalent double-dispatch Java code. By translating directly into compiled code, they apply a textual priority to avoid the thorny issue of ambiguous methods. Unfortunately, the parasitic method selection process is a sequence of several dispatches to search over a potentially exponential tree of overriding methods.

The language extension and preprocessor approach has other limitations. First, existing tools do not support the extensions; for example, debuggers do not elide the automatically generated double-dispatch routines. Second, instance methods appear to only take arguments that are objects, which is too limiting. Our experience with Swing shows that existing programs often double dispatch on literal `null` and array arguments and pass primitive types as arguments; multi-methods need to support these non-object types. Third, preprocessors limit code reuse and extensibility; adding multi-methods to an existing behaviour requires either access to the original source code or additional double-dispatch layers.

Chatterton [8] examines two different multi-dispatch techniques in mainstream languages: C++ and Java. First, he considers providing a specialized dispatcher class. Each class that participates as a method receiver must register itself with the dispatcher. To relieve the programmer of this repetitive coding process, he provides a preprocessor that rewrites the Java source to include the appropriate calls. Each method, marked with the keyword *multi*, is also expanded by the preprocessor into many individual methods, one for each combination of classes (and superclasses). A method invocation is replaced by a call to the dispatcher which searches via reflection for an exact match. That method is then invoked. This system suffers from exponential blowup of methods.

Chatterton's second approach examines the performance of various double dispatch enhancements. He provides a modified C++ preprocessor which analyses the entire Java program. It can build a number of different double-dispatch structures, including cascaded and nested `if...else-if...else` statements, inline `switch` statements, and simple two-dimensional tables. Again, he expands every possible argument-type combination in order to apply fast equality tests rather than slow subtype checks. A significant restriction is that full-program analysis is required. This defeats the ability to use existing libraries and diminishes Java's dynamic class loading benefits.

One interesting language for multi-dispatch is Leavens and Millstein's *Tuple* [18]. They describe a language "similar in spirit to C++ and Java" that permits the programmer to specify at each call-site the individual arguments that will be considered for multi-dispatch. This paper does not describe an implementation; it appears to be a model of potential syntax and semantics only. A future project might be to implement his syntax specifically into the Java environment. In particular, a simple syntax extension would allow `super` method invocations on arbitrary multi-dispatch arguments.

Another recent development is *MultiJava* [11]. There, the authors extend the Java language with additional syntax to support open classes and multi-dispatch. The MultiJava compiler emits double-dispatch type-case bytecodes for invocations of the open-class methods and multi-methods. The emitted bytecode is accepted by standard JVMs, but suffers a substantial overhead from interpreting slow subtype-testing bytecodes. Unfortunately, multi-dispatch can only apply to methods defined using the open-class syntax and only within the program text that imports the open-class definitions. If subclasses wish to further specialize the multi-methods, additional open-class definitions are required. Compilation of these further open-subclasses may result in multiple layers of type-case double-dispatch. Internally, MultiJava inlines the multi-method bodies into a static method in a separate anchor class – this means that the multi-methods disappear from the binary code and become invisible to the reflective subsystem in Java. Finally, MultiJava is a paper design at this time[16], so performance comparisons are not possible.

[16] Personal communication at OOPSLA 2000.

9 Concluding Remarks

We have presented the design and implementation of an extended Java Virtual Machine that supports multi-dispatch. This is the first published description of how to implement arbitrary-arity multi-dispatch in Java. In contrast to the more verbose and error-prone double-dispatch technique, currently found in the AWT (Figure 2), multi-dispatch typically reduces the amount of programmer-written code and generally improves the readability and level of abstraction of the code.

Our approach preserves both the performance and semantics of the existing dynamic uni-dispatch in Java while allowing the programmer to select dynamic multi-dispatch on a class-by-class basis without any language or compiler extensions. The changes to the JVM itself are small and highly-localized. Existing Java compilers, libraries, and programs are not affected by our JVM modifications and the programs can achieve performance comparable to the original JVM (Table 2).

In a series of micro-benchmarks, we showed that our prototype implementation adds no performance overhead to dispatch if only uni-dispatch is used (Table 2) and the overhead of multi-dispatch can be competitive with explicit double dispatch (Table 4).

We have also introduced and implemented an extension of the Java Most Specific Applicable (MSA) static multi-dispatch algorithm for dynamic multi-dispatch. In addition, we have performed the first head-to-head comparison of table-based multi-dispatch techniques implemented in a mainstream language. In particular, we implemented Single Receiver Projections (SRP). Overall, our tuned SRP implementation performs as well (or better) than programmer-targeted multi-dispatch. With performance improvements in concurrency, we expect our tuned system to out-perform type-case double dispatch.

References

[1] E. Amiel, O. Gruber, and E. Simon. Optimizing multi-method dispatch using compressed tables. In *OOPSLA 1994 Conference Proceedings*, pages 244–258. Association for Computing Machinery, October 1994.

[2] P. Andre and J. Royer. Optimizing method search with lookup caches and incremental coloring. In *OOPSLA 1992 Conference Proceedings*. Association for Computing Machinery, 1992.

[3] J. Boyland and G. Castagna. Parasitic methods: An implementation of multi-methods for Java. In *OOPSLA 1997 Conference Proceedings*, pages 66–76. Association for Computing Machinery, November 1997.

[4] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. T. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[5] T. Budd. *An Introduction to Object Oriented Programming, Second Edition.* Addison-Wesley, 1997.

[6] C. Chambers. Object-oriented multi-methods in Cecil. In *ECOOP 1992 Conference Proceedings*, pages 33–56. Springer-Verlag, June 1992.

[7] C. Chambers and W. Chen. Efficient multiple and predicate dispatching. In *OOPSLA 1999 Conference Proceedings*, pages 238–255. Association for Computing Machinery, November 1999.

[8] D. Chatterton. *Dynamic Dispatch in Existing Strongly Typed Languages.* PhD thesis, School of Computing, Monash University, Monash, Australia, 1998.

[9] W. Chen. Efficient multiple dispatching based on automata. Master's thesis, GMD-ISPSI, Darmstadt, Germany, 1995.

[10] W. Chen, V. Turau, and W. Klas. Efficient dynamic lookup strategy for multi-methods. In *ECOOP 1994 Conference Proceedings*, pages 408-431. Springer-Verlag, July 1994.

[11] C. Clifton, G. T. Leavens, C. Chambers, and T. Milstein. MultiJava: Modular symmetric multiple dispatch and extensible classes for Java. In *OOPSLA 2000 Conference Proceedings*, pages 130–145. Association for Computing Machinery, October 2000.

[12] E. Dujardin, E. Amiel, and E. Simon. Fast algorithms for compressed multimethod dispatch table generation. *ACM Transactions on Programming Languages and Systems*, 20(1):116–165, January 1998.

[13] C. Dutchyn. Multi-dispatch in the *Java Virtual Machine*: Design and implementation. Master's thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 2001. In preparation.

[14] A. Goldberg and D. Robson. *Smalltalk-80 The Language and its Implementation.* Addison-Wesley, 1983.

[15] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, 2nd Edition.* Addison-Wesley, 2000.

[16] W. Holst, D. Szafron, Y. Leontiev, and C. Pang. Multi-method dispatch using single-receiver projections. Technical Report 98-03, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1998.

[17] W. M. Holst. *The Tension between Expressive Power and Method-Dispatch Efficiency.* PhD thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 2000.

[18] G. T. Leavens and T. D. Millstein. Multiple dispatch as dispatch on tuples. In *OOPSLA 1998 Conference Proceedings*, pages 244–258. Association for Computing Machinery, October 1994.

[19] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *OOPSLA 1998 Conference Proceedings*, pages 36-44. Association for Computing Machinery, October 1998.

[20] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, 2nd Edition.* Addison-Wesley, 1999.

[21] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and Y. Kimura. OpenJIT: An open-ended, reflective JIT compile framework for Java. In *ECOOP 2000 Conference Proceedings.* Springer-Verlag, 2000.

[22] C. Pang, W. Holst, Y. Leontiev, and D. Szafron. Multiple method dispatch using multiple row displacement. In *ECOOP 1999 Conference Proceedings*, pages 304--328. Springer-Verlag, June 1999.

[23] G. L. Steele. *Common Lisp.* Digital Press, 1985.

[24] B. Stroustrup. *The C++ Programming Language: Third Edition.* Addison-Wesley, 1997.

[25] J. Vitek and R. N. Horspool. Compact dispatch tables for dynamically typed programming languages. In *Proceedings of the International Conference on Compiler Construction*, 1996.

[26] K. Walrath and M. Campione. *The JFC Swing Tutorial: A Guide to Constructing GUIs.* Addison-Wesley, 1999.

Using Accessory Functions to Generalize Dynamic Dispatch in Single-Dispatch Object-Oriented Languages

David Wonnacott
Department of Computer Science
Haverford College
Haverford, PA 19041
davew@cs.haverford.edu
http://www.cs.haverford.edu/people/davew

Abstract

Object oriented languages generally include some form of *dynamic dispatch*; that is, in the absence of precise compile-time type information, they perform a run-time selection of the appropriate function body (or *method*) from a set of candidates. Existing single-dispatch languages restrict dynamic dispatch to the object receiving the message.

Such languages exhibit a conflict between the goals of providing an extensible a set of types and providing an extensible the set of operations that can be performed on these types. We show that this conflict is a consequence of the restriction of dynamic dispatch to the receiver object. We also demonstrate that this conflict can be resolved by introducing a generalized form of single dispatch (thus avoiding the complexity of multiple dispatch). On this evidence, we argue that dispatch technique should be decoupled from membership in a class and access to its representation.

1 Introduction

Inheritance helps a programmer create a set of classes for distinct yet similar types of objects. When inheritance is used for subtyping [13], the superclass lists the messages that must be handled by its subclasses. Code that uses superclass messages will work for subclass objects, as long as these objects respond properly to all messages listed in the superclass. Such code will also work without modification (or even recompilation) with objects of subclasses that are added to the system later.

This ability to reuse code as new kinds of objects are added to the system is touted as one of the major advantages of the object-oriented approach. However, it comes at the expense of our ability to reuse code as new operations are added to a system. Consider the general problem of designing software in which various interpretations (i.e. methods or functions) are defined for various kinds of objects (i.e. classes), as discussed by Harrison and Ossher [10], Krishnamurthi, Felleisen, and Friedman [11], and Appel [2, Section 4.2]. For example, Appel focuses on the design of an abstract syntax tree (A.S.T.) for a compiler: The different kinds of objects correspond to various program structures including expressions, statements, and declarations, while the different interpretations include static checks (such as type checking), various optimizations, or code generation for various architectures.

Such a system can be designed in the traditional object-oriented style, which lets us add new kinds of objects. However, new interpretations must be added to the superclass, requiring modification of existing source code and recompilation. The need to add operations to the class also violates a basic principle of data encapsulation, that each class should be defined with a minimal set of operations and edited only for a redesign, not a reuse.

We could, of course, abandon the object-oriented style, and adopt a style in which each function contains code for every type of object it could operate on. This lets us add new interpretations, but the introduction of a new kind of object forces us to edit each of the existing functions. We have once again prevented encapsulation and reuse without recompilation, and lost other benefits of object-oriented style as well. Appel argues that the latter style is more appropriate for his A.S.T. example, while

the former is more appropriate for classic object-oriented systems (such as graphical user interfaces).

We demonstrate that this choice of styles represents a limitation of traditional object-oriented languages, not a fundamental design choice. Specifically, it is a consequence of the restriction of dynamic dispatch to the methods listed in a class. If we relax this restriction, we can create a system in which existing code can be reused (without access to source code) as both new kinds of objects and new interpretations are added. We show that it is possible to allow dynamic dispatch for functions not listed in a class (which we call *accessory functions* of the class), and demonstrate that accessory functions can be implemented efficiently.

This paper is organized as follows: We begin, in Section 2, with a brief review of the use of dynamic dispatch and its impact (both positive and negative) on code reuse. This section also covers the implementation of dynamic dispatch in C++. In Section 3, we spell out the goals that we intend to achieve by generalizing dynamic dispatch, describe the semantics (and C++ syntax) for accessory functions, and show that accessory functions can be used to enhance reuse in our example program. We then discuss, in Section 4, the relationship of accessory functions to other language properties such as support for data encapsulation. In Section 5, we briefly discuss the implementation of accessory functions in C++. Finally, we discuss related work in Section 6, and give our conclusions in Section 7.

2 Dynamic Dispatch and Reuse

One argument made by advocates of object-oriented languages is that object-oriented programming can facilitate code reuse. A well-designed class or hierarchy of classes can be reused without knowledge of, or access to, its implementation (just as a well-designed function or procedure can be reused in other languages). In this paper, we will be concerned with two types of reuse of classes. In the first, which we call reuse by inheritance, a programmer represents a new kind of data by making an extension of some existing data type. In the second, which we call reuse in a function, a programmer uses a class in the implementation of a new function (perhaps for a local variable or parameter).

We will focus on reuse that can be accomplished without modification of the source code that is to be reused. This is obviously important if software is distributed without source code, or if programmers are not able to modify the source code. It also prevents unnecessary code management complexities when source code is available. If several groups of programmers each reuse the same class, their modifications to that class must be merged, and the merged code must be tested, if these extensions are ever to be used together.

Note that reuse by inheritance (as defined above) can occur even in languages that do not support inheritance directly. Consider for example the task of representing various kinds of expression nodes in a compiler's abstract syntax tree (A.S.T.). (This example was adapted from [2]; we have focused on a simple method that can be understood with minimal knowledge of compiler construction.) In languages like C, the programmer can use a single `struct` to represent all kinds of expression nodes, distinguishing among the different kinds of nodes by including a `kind` field in the `struct` and using a `switch` statement to select code that is appropriate for each kind of expression node. Reuse by inheritance occurs if the programmer adds a new kind of expression node (perhaps because a new kind of expression has been added to the language). However, this reuse requires access to (and modification of) the existing code – the programmer must add a new `case` to the `switch` statements in existing functions, even if existing `cases` do not need to be modified.

In an object-oriented language like C++, the programmer can define the original kinds of expression nodes with a collection of classes, each of which inherits from an "abstract superclass" `Exp` (shown in C++ in Figure 1). The abstract superclass gives methods that are shared by all kinds of expressions, such as a `print_rep` method to produce a string that gives a printable representation for any kind of expression node. These methods must be defined for every subclass of `Exp`, and we can therefore request the printable representation of any object denoted by a reference of type `Exp &` (which must be of a class derived from `Exp`). Methods that are specific to one kind of node are defined only in the appropriate subclass (and thus cannot be requested through references of type `Exp &` in statically checked languages like C++).

We rely on the fact that we can request the `print_rep` for any object referred to by an `Exp &` in

```
class Exp {  // abstract superclass
public:
  virtual string print_rep() = 0;
private:
  ...
};

class Num : public Exp {
public:
  int value();
  virtual string print_rep();
private:
  ...
};

string Num::print_rep()
{
  // convert integer
  // to ASCII string
  return itoa(value());
}

class Plus : public Exp {
public:
  Exp &lhs();
  Exp &rhs();
  virtual string print_rep();
private:
  ...
};

string Plus::print_rep()
{
  return
    ''('' + lhs().print_rep() +
    ''+'' + rhs().print_rep() + '')'';
}
```

Figure 1: Dynamic Dispatch Example

the `print_rep` method for class `Plus`. This method uses the printable representation for the left and right operands of the sum. Dynamic dispatch ensures that the `print_rep` method for the correct class is used, even though the compiler cannot determine statically which method will be chosen. This approach lets the programmer add new kinds of nodes by deriving a new class (with an appropriate `print_rep`) for each new kind of node. This does not require any modification of existing source code, and dynamic dispatch will ensure that the new class's `print_rep` is used for the new nodes, even in existing code (such as the `print_rep` method for class `Plus`).

Dynamic dispatch shifts the responsibility of selecting the appropriate `print_rep` code from the programmer to the programming language. In single-dispatch languages like C++ and Java, dynamic dispatch can be implemented by associating, with each object, a table of pointers to the code for each of the object's methods. In our example, each object of a class that inherits from Exp has a pointer to its `print_rep` method at a fixed offset in its dispatch table – to perform a call to `print_rep` when the type of object is not known, the compiler can generate an indirect function call using the table.

Note that it is possible for the programmer to implement dynamic dispatch in non-object-oriented languages that allow pointers to functions, such as C. However, this requires that the programmer undertake the tedious and potentially error-prone task of initializing using the tables of function pointers.

Unfortunately, the use of inheritance and dynamic dispatch inhibits reuse in a function. This is a consequence of the fact that only methods listed in a class can be dynamically dispatched based on that class. Consider what would happen if we wish to add a new pass to our compiler (such as an operation to interpret an expression), rather than a new type of expression node. If we had used a single class with a `kind` field, we could simply create a new function that takes an expression node, checks the `kind` field, and interprets the node in the appropriate way. However, if we wish to add this operation to the collection of classes in Figure 1, me must edit the class definitions to add an `interpret` method.

Editing the existing class definitions would be appropriate if we were redesigning, rather than reusing, these classes. However, we do not believe that every new use that requires dynamic dispatch

should be considered a redesign: If this were the case, the author of a class would have the responsibility of enumerating all cases in which dynamic dispatch is needed for objects of that type.

There are a number of other ways to add an `interpret` method, each of which we consider unsatisfactory. First, we could introduce an `interpret` function that is not part of the `Exp` classes (in Java, it must be a method of some other class, such as a class `Interpreter`). This function could use `typeid` (in C++) or `instanceof` (in Java) in a series of `if` statements to select the appropriate code for the kind of node being interpreted. In a language without an equivalent of `typeid`, the programmer can add a `kind` field (or operation). In either case, this approach simply sets up a future problem with reuse by inheritance – any programmer adding a new kind of expression node must edit the code for `interpret` to work with the new type.

We could also use inheritance to produce new subclasses that add an `interpret` operation (deriving a class `Exp_with_interpret` from class `Exp`). However, this introduces spurious uses of multiple inheritance, which we consider highly undesirable (though we have no problem with legitimate uses of multiple inheritance): If we are to apply `interpret` to each new kind of node through an `Exp` reference, then the new node classes must share a common superclass with an `interpret` function, which introduces a second superclass for the new node classes.

Thus, if dynamic dispatch is provided only to functions listed in the class, we are forced to choose between allowing reuse in functions (if we use an explicit `switch`) or reuse by inheritance (if we use dynamically dispatched methods). To allow both kinds of reuse, we must allow dispatch for functions outside of the class. This is allowed in some languages that provide *multiple dispatch* [3, 9]. However, multiple dispatch has a higher run-time cost than single dispatch: techniques based on complete dispatch tables may require large tables, and other methods do not provide constant-time dispatch [1]. Accessory functions provide both kinds of reuse without the added complexity and cost of multiple dispatch.

Multiple dispatch can also be introduced as a programming technique rather than a language feature (for example, by using the visitor design pattern). This also introduces unnecessary overhead, and is less flexible than a compiler-generated dispatch, as we will see in Section 6.

```
// "pure virtual" accessory function
// for superclass
int interpret(virtual Exp &) = 0;

int interpret(virtual Num &n)
{
   return n.value();
}

int interpret(virtual Plus &p)
{
   return interpret(p.lhs())
        + interpret(p.rhs());
}
```

Figure 2: Accessory Function Example

3 Accessory Functions

Although no current single-dispatch language does so, it is in principle possible to allow dynamic dispatch on a parameter other than the receiver of the message. We call a function that does so an *accessory function* of the class involved in dispatch. Figure 2 shows how accessory functions can be used to add a dynamically dispatched `interpret` function to our A.S.T. example of Figure 1 (using a notation based on C++). The rest of this section gives our design goals, and gives possible syntax and semantics for integrating accessory functions into a C++ -like language.

Our goal is to provide the following properties of programs written with accessory functions:

- Accessory functions can be added to a group of classes without editing (or even reading) the source for the classes. To avoid violated the principle of data encapsulation, accessory functions do *not* have access to the private data of any classes they are not listed in. For example, the `interpret` functions of Figure 2 do not have access to the private data of any class, including the classes for the abstract syntax tree.
- New classes can be added to a program that uses accessory functions without any need to edit existing functions or classes. In other

words, we wish to allow both reuse in a function (the previous item) and reuse by inheritance.

- Accessory functions can be dispatched as efficiently as other single dispatch functions, such as virtual functions in C++.
- Except for the change in which argument is used for dynamic dispatch, function dispatch should follow the rules that exist in the language.
- The system must be able to produce errors about dispatch before the program is executed: A user must not see "method not found" errors while running a program (this was a design goal of C++).

3.1 Syntax

We need syntactic mechanisms to identify the parameter to be used in dynamic dispatch and to specify that a superclass function should be selected during a call in a subclass function. In this article, we give a syntax that is an extension of C++, and focus on definitions that are appropriate for C++, though accessory functions could be added to other statically typed single-dispatch object-oriented languages.

We identify an accessory function by using the keyword `virtual` in the declaration of a parameter. We consider `virtual` to be an attribute of a parameter rather than an attribute of the function itself. When `virtual` is used in the traditional way, we say that the member function has a virtual receiver (rather than a virtual parameter). Accessory functions for C++ may be created outside of any class, as in Figure 2, or they may be created as members (or friends) of one (or more) classes.

When an accessory function for a subclass needs to make use of the superclass function, it gives explicit type information for the virtual parameter. We use syntax that is similar to type casting for this purpose (we chose this notation because it produces the result that type casting of a reference produces for a statically dispatched function). To avoid introducing a new keyword, we reuse the word "virtual" for this purpose, i.e. the `interpret` function for `Num` could call the superclass function (were it not pure virtual) with the syntax `interpret( (virtual Exp) n )`. This is only legal if the new type is a public superclass of the argument type; its effect is analogous to using `interpret( (Exp &) n )` for a statically dispatched function.

3.2 Restrictions

We place several restrictions on the definition of accessory functions. Most are needed to prevent ambiguities that prevent us from selecting between dynamic and static dispatch at compile time.

- For any function, at most one parameter (including the receiver object) may be virtual. This is necessary to ensure that we do not need multiple dispatch. Here and in the remainder of this paper, we count the receiver object of a method as a parameter.
- No single scope can contain two functions that differ only in the dispatch mechanism of a parameter: We cannot have `f(Exp &)` and `f(virtual Exp &)`. This restriction is necessary because it would not be possible to distinguish calls to the two functions. C++ has an analogous rule for virtual functions.
- In any one scope, no two functions with the same name and arity (number of arguments) are dynamically dispatched on different parameters. This will play an important role in our function selection semantics below.
- All functions with parameters of class C must be defined before the execution of code that creates an object of class C. In traditional C++ environments, all functions are defined before program execution begins, and this restriction always holds. In environments that allow dynamic loading of classes (such as Java) this places restrictions on the relative timing of object creation and the loading of functions.
- To ease implementation in C++, we only allow accessory functions for classes that already have at least one virtual function: For example, we cannot have `f(virtual int &)`, as `int` has no dispatch table.

3.3 Function Selection Semantics

Function dispatch based on the types of multiple arguments, whether static or dynamic, raises two

challenges: We must specify which function body is considered the correct choice for any given call, and we must provide a way for the program to branch to this code efficiently. In this section, we consider the question of how to adapt the existing dispatch rules of C++ for accessory functions, leaving the the question of how to branch to this function for Section 5.

The traditional choice of static vs. dynamic dispatch, and the new decision of which argument is to be used for dynamic dispatch, must be made at compile-time. These decisions are thus based on the types of the references used in the call (rather than the types of the objects they refer to), and the set of functions that are in scope at the point of the call. Once the compiler has selected dynamic dispatch on a particular object, the true "run-time" type of the object will be used in the actual call.

We ensure that we can statically determine which argument is to be used in dynamic dispatch by requiring that, in any one scope, no two functions with the same name and arity (number of arguments) are dynamically dispatched on different parameters. Essentially, we consider dispatch mechanism to be an attribute of the *message* (function name) rather than *method* (function body). Conflicts that might arise when two independent projects happen to use the same function names must be resolved via namespaces.

This restriction allows us to use the traditional C++ approach to dispatch: We select, from the set of functions that are in scope, the one with parameters types that best match the compile-time type information about the arguments used in the call. If there is no unique best match, we generate an error message. We then generate either a dynamic dispatch (based on the appropriate parameter type, if one parameter is virtual) or static dispatch (if no parameter is virtual).

Thus, if a group of functions of a given name and arity are dispatched on argument a, we produce a branch to the function that would have been called if all functions with this name and arity had been written as (possibly virtual) member functions of the classes of their a^{th} arguments. In other words, we generate a branch to the function that would have been called if we had violated the encapsulation of the classes.

As we will see in Section 5, our implementation allows us to produce warnings for certain surprising behavior that is a consequence of this combination of static and dynamic information.

3.4 Type Casting

The compiler will not produce a virtual argument by applying an implicit type cast to a value (though it may still convert a subclass type reference (or pointer) to a superclass reference (or pointer)). This is an extension of the existing C++ rule that the compiler will not apply an implicit cast to produce the receiver object. `Virtual` is generally used in the declaration of a pointer or reference type parameter: When applied to the declaration of a value parameter, it affects type casting, but not dispatch (since complete type information must be present at compile time).

The lack of casting for virtual arguments means that adding `virtual` to a parameter of an existing function may interfere with the compilation of code that had used this function: It may be necessary to add an explicit cast where an implicit one had been used previously to produce the (non-virtual) argument. We believe it would be possible to implement a system that allows implicit casting for accessory functions, but that such a system could produce highly confusing results, as casting is based on the functions that are in scope, but dispatch is based on all compatible functions in the final program.

3.5 Default Arguments

The above discussion ignores the issue of default arguments in C++. We believe these can be handled by treating a declaration with a default argument as if it were a group of declarations of overloaded functions, all but one of which simply supply extra arguments and call the original function.

4 Encapsulation

Existing single-dispatch object-oriented languages link together the following three properties: (1) The class(es) in which a function is defined, (2) The class(es) representation(s) that a function can ac-

cess, and (3) The class that is used in dynamic dispatch of the function. In many languages, these three properties unified in the concept of "the" class of a method. C++ allows slightly more flexibility by allowing a function to access the representations of several classes if it is listed as a friend (or member) in each of them, though it can only be dynamically dispatched based on the (single) class of which it is a member. Even some multiple-dispatch languages unify the concept of access and dispatch: For example, in Cecil "a multi-method is granted privileged access to all objects of which the multi-method is a part, i.e., of the objects that are the method's argument constraints" [5, Section 1.5].

The unification of properties (1) and (2) essentially defines data encapsulation, which plays an essential role in reuse of classes. Since direct access to a class's representation is allowed only from those functions included in the class itself, we can rest assured that uses of the class by other functions (including all "reuse") will not corrupt any properties guaranteed by the class as it was originally written. Implicit in the idea of data encapsulation is the principle that programmers will not rampantly add operations to a completed class. If new operations are added to a class, and thus granted access to its representation, we can no longer guarantee that the representation cannot be corrupted. To retain this important property, accessory functions do *not* have access to classes involved in their dispatch (unless they are listed as friends of that class, for some reason).

We have proposed that the property of dynamic dispatch be separated from the property of inclusion in (and access to) a class. While we originally argued that this be done to support reuse, we find it appealing for several other reasons. First, it provides greater orthogonality of language features. Properties (1) and (2) above must remain unified, but dynamic dispatch is now fully independent. Second, we believe that accessory functions strengthen language support for data encapsulation. One tenet of data encapsulation is that each class should be defined with a set of operations that is both *adequate* and *minimal*:

> There can also be too many operations in a type... In this case, the abstraction may be less comprehensible, and implementation and maintenance are more difficult. The desirability of extra operations must be balanced against the cost of implementing these operations. If the type is adequate, its operations can be augmented by procedures that are outside the type's implementation. [14, Section 4.9.3]

Stroustrup also discusses this principle [17, Section 11.5.2]. Thus, it can be argued that both the `interpret` and `print.rep` functions belong outside the A.S.T. classes in our motivating example, as both can be written efficiently in terms of existing operations. However, without accessory functions, these operations must be placed inside the class.

Note that our need for dynamic dispatch for our A.S.T. example is not simply an artifact of the fact that we have not provided a more abstract way of traversing an abstract syntax tree. If we provide either an iterator or a traversal function to apply arbitrary code to each element of the tree, we still find the need to associate certain code with certain kinds of A.S.T. nodes. Dynamic dispatch provides a simple and efficient mechanism for doing so. Only the restriction of dispatch to members of the class keeps us from using it in these cases.

Since accessory functions do not have access to the private data of the data structure to which they are applied, they cannot save state information in this structure. We must, therefore, accumulate any needed information in some other way. In our "interpret" example, information is kept as temporary values in the C++ run-time stack; this works because we only need to produce a single final value (the result of the expression). If we need more complex information, such as a value associated with each node in the tree, we can build up an auxiliary data structure (for example, a second tree that contains values that correspond to the nodes in the A.S.T.). If we wish to traverse the data structure and modify it, we must modify the class (by using traditional virtual functions instead of accessory functions, or making the accessory functions into friends of the class). This is consistent with the principle that only operations listed in the class can access the class's private data.

5 Implementation for C++

Given the definitions and rules of Section 3, the implementation of accessory functions does not pose many interesting technical challenges. We simply

move the existing algorithms for building dispatch tables from compile-time to link-time, retaining the general principles used for virtual functions in C++: each object contains a pointer to a table of all functions that may be dynamically dispatched based on its type, and the compiler statically produce code that will locate a given function with a (constant time) table lookup (for single inheritance, we simply use a fixed offset into the table).

The restrictions in Section 3.2 can be checked trivially as function declarations are processed at compile-time. To compile a (possibly dynamically dispatched) function call, we start by applying the C++ rules for overloaded function selection [17, Section 7.4] to the set of functions that are in scope and have the correct name and number of parameters, with the restriction that type casting of values cannot be used to create a match with a virtual parameter. If there is not a unique match, a compile-time error is produced. If there is a unique best match, we generate either a regular function call (if the best match has no virtual parameter) or a dynamic function dispatch.

If the best match had a virtual parameter, the dynamic dispatch is very much like a C++ virtual function call. We know that the virtual argument will contain a pointer to a table of dynamically dispatched functions, and generate a load of a function pointer from this table, and a branch to this address. The presence of accessory functions means that we no longer know the size or layout of this table when compiling a single file, but we handle this by treating the offset into the table as an undefined reference that will be filled in later by the linker.

Note that we need a separate entry in the dispatch table for each virtual parameter position, function name, and arity. A group of three-parameter functions may be dispatched differently from some two-parameter functions of the same name; different three-parameter functions may have different virtual parameters (as long as they are not in the same scope).

We rely on the compiler to give the linker a complete description of the DAG describing the class inheritance structure, and a list of all functions (complete with parameter types and information about which parameters are virtual). We topologically sort the inheritance DAG and we apply the algorithm used to build C++ virtual function tables, using a new offset for each set of statically distinct functions. Since the offsets are determined at this stage, we can resolve the undefined references produced at compile time.

This implementation places an increased load on the linker, and thus may increase link times. However, this is a fundamental consequence of the fact that declarations of accessory functions for a given class may be spread across several files (unlike the class' virtual functions), not a weakness of our implementation. The distribution of accessory functions over different files prevents detection by the compiler of certain errors that could be detected by traditional C++ compilers, such as the instantiation of a class with a pure virtual accessory function (one file may instantiate an object of a class for which pure virtual accessory functions are created in another file).

It may be possible to reduce the link-time overhead somewhat by replacing our implementation with a version of Millstein and Chambers' techniques for modular multimethod dispatch [15], restricted to the case of single dispatch. We have not investigated this possibility, since some degree of link-time overhead is unavoidable, and our main goal is to present a simple implementation that demonstrates that we can retain the constant-time nature of the dynamic dispatch used in C++.

6 Related Work

Snyder [16] and Liskov [13] have also studied conflicts between data encapsulation and other aspects of object-oriented programming. They discuss problems that arise when subclass operations are given access to superclass data or private operations, and Snyder [16] observes that a class's superclasses cannot be considered an implementation detail of the class in a system that allows multiple inheritance without replicating common superclasses.

Appel [2, Section 4.2], Harrison and Ossher [10], Krishnamurthi, Felleisen, and Friedman [11], and possibly many others have noted the conflict that arises between reuse by inheritance and reuse in a function. There are a number of approaches to resolving this problem, which we discuss in order of increasing familiarity for programmers familiar with C++.

Some techniques for multiple dispatch (also known

as multi-methods) [3, 9, 5, 4] could be used to provide dispatch on a parameter other than the receiver of an object, or by including the type needed for dispatch in a new tuple type [12]. However, general multi-method dispatch either requires more than constant time per dispatch or excessively large dispatch tables [1]. However, recent techniques for multimethod dispatch [6] have very low overhead, and we believe they would be at least as efficient as our system for the case of single dispatch (which, as we have noted, is all that is needed to resolve the conflict between different kinds of reuse).

Work on multiple dispatch also differs from ours in that it not focused on the separation of dispatch and access. Cecil explicitly retains the unification of dispatch and access, though a change to this rule would probably not have any impact on performance. We believe the main barrier to the widespread use of these techniques to enable both kinds of reuse is the tendency of programmers to prefer familiar techniques and languages. The other approaches to solving this problem (including ours) focus on techniques or language extensions that can be applied to C++ or Java. General discussions of techniques for multi-method dispatch can be found in [1, Section 3.2] and [6, Section 3.7].

The visitor pattern could be applied to our abstract syntax tree example: Each tree class (such as `Plus`) would provide a "visit" operation that takes a "visitor" parameter, and sends the visitor a message that is specific to the tree subclass (e.g. `Plus::visit(visitor &v)` sends `v.visitPlus(this)`). This approach still interferes with the addition of new subclasses, since the visitor class must be extended to include a new method for each new subclass. The "Extended Visitor" protocol [11] fixes this problem, but still has higher overhead than a single dispatch accessory function, and to some degree shifts the burden of performing dispatch back from the compiler onto the programmer. It thus creates unnecessary opportunities for programmer error, and suffers from limitations due to the lack of compiler support. Krishnamurthi, Felleisen, and Friedman have developed a language named Zodiac to simplify the use of the extended visitor pattern, but it is not clear how quickly it will be adopted by programmers who are familiar with C++ or Java.

Harrison and Ossher [10] proposed the "Subject-Oriented Programming" style. This approach, like our accessory functions, can serve as the basis for extension of an existing language like C++ (it is currently available as a preprocessor for C++ in IBM's Visual Age for C++ Version 4). Instead of separating the property of dispatch from presence in a class, subject-oriented programming facilitates the decomposition of a class into different "subjects" that can be developed independently and then composed. A subject can correspond to one of our accessory functions, a group of functions, or functions together with associated data (like a class). This approach is more general than ours (though not more general than some of the multimethod systems), and correspondingly raises more new issues for programmers, such as the selection of composition system.

We have focused on providing a resolution to the conflict between reuse by inheritance and reuse in a function, while creating the minimal impact on programmers who are familiar with the traditional object-oriented style. Our extensions can be added to C++ by relaxing a single rule (that dynamic dispatch must be based on the receiver of the message). A preliminary description of accessory functions appeared at MASPLAS '99 [7]. We have also explored the possibility of allowing multiple virtual parameters [8], though this work does not make a significant contribution to the existing literature on multiple dispatch.

7 Conclusions

Current single-dispatch object-oriented languages provide dynamic dispatch only for functions listed in the class involved in dispatch, even if overloading is allowed for other parameters. This property gives the author of a class the responsibility of enumerating all cases in which dynamic dispatch is needed for objects of this type. This hinders code reuse by forcing the designer of a set of types to choose between allowing reuse by inheritance (by using dynamic dispatch) and reuse in a function (by using explicit `switches` on the kind of object).

It is possible and (we believe) desirable to provide dynamic dispatch to users of a class hierarchy. In other words, we should eliminate the coupling between dispatch method and membership in (and access to) a class. This decoupling lets programmers achieve both reuse by inheritance and reuse in a function. Thus, our "accessory functions" improve the support for both reuse and data encapsulation,

and can be implemented with the same efficient dispatch algorithms used in current C++ virtual function selection.

8 Acknowledgments

This work is supported by NSF grant CCR-9808694.

References

[1] E. Amiel, O. Gruber, and E. Simon. Optimizing multi-method dispatch using compressed dispatch tables. In *OOPSLA'94 – Object Oriented Programming Systems, Languages and Applications*, pages 244–258, Oct. 1994. Published as SIGPLAN Notices Vol. 29, No. 10.

[2] A. W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[3] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. Commonloops: merging lisp and object-oriented programming. In *OOPSLA'86 – Object Oriented Programming Systems, Languages and Applications*, pages 17–29, 1986.

[4] J. Boyland and G. Castagna. Parasitic methods: an implementation of multi-methods for java. In *OOPSLA'97 – Object Oriented Programming Systems, Languages and Applications*, pages 66–76, Oct. 1997. Published as SIGPLAN Notices Vol. 32, No. 10.

[5] C. Chambers. Object-oriented multi-methods in cecil. In *ECOOP '92 Conference Proceedings*, July 1992.

[6] C. Chambers and W. Chen. Efficient multiple and predicate dispatching. In *OOPSLA'99 – Object Oriented Programming Systems, Languages and Applications*, pages 238–255, Oct. 1999. Published as SIGPLAN Notices Vol. 33, No. 10.

[7] C. B. Flynn and D. Wonnacott. Encapsulation, extension, and function dispatch in C++. In *The 1999 Mid-Atlantic Student Workshop on Programming Languages and Systems (MASPLAS '99)*, Apr. 1999.

[8] C. B. Flynn and D. Wonnacott. Reconciling encapsulation and dynamic dispatch via accessory functions. Technical Report DCS-TR-387, Dept. of Computer Science, Rutgers U., June 1999. Available as ftp://www.cs.rutgers.edu/pub/technical-reports/dcs-tr-387.ps.Z.

[9] J. Guy L. Steele. *Common Lisp: The Language (second edition).* Digital Press, 1990.

[10] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *OOPSLA'93 – Object Oriented Programming Systems, Languages and Applications*, Sept. 1993. Published as SIGPLAN Notices Vol. 28, No. 10.

[11] S. Krishnamurthi, M. Felleisen, and D. P. Friedman. Synthesizing object-oriented and functional design to promote reuse. In *Proceedings of the Twelfth European Conference on Object-Oriented Programming ECOOP '98*, Apr. 1998.

[12] G. T. Leavens and T. D. Millstein. Multiple dispatch as dispatch on tuples. In *OOPSLA'98 – Object Oriented Programming Systems, Languages and Applications*, pages 374–387, Oct. 1998. Published as SIGPLAN Notices Vol. 33, No. 10.

[13] B. Liskov. Data abstraction and hierarchy (keynote address). In *OOPSLA'87 – Object Oriented Programming Systems, Languages and Applications (Addendum)*, pages 17–34, Oct. 1987. Published as SIGPLAN Notices Vol. 23, No. 5.

[14] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* The MIT Press, Cambridge, Mass., 1986.

[15] T. Millstein and C. Chambers. Modular statically typed multimethods. In *Proceedings of the Thirteenth European Conference on Object-Oriented Programming ECOOP '99*, pages 279–303, June 1999. Published as Springer-Verlag LNCS 1628.

[16] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPSLA'86 – Object Oriented Programming Systems, Languages and Applications*, pages 38–48, Oct. 1986.

[17] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 1997.

The Design and Performance of Meta-Programming Mechanisms for Object Request Broker Middleware

Nanbor Wang Kirthika Parameswaran Douglas Schmidt Ossama Othman

{nanbor, kirthika}@cs.wustl.edu
Department of Computer Science
Washington University, St.Louis

{schmidt, ossama}@uci.edu
Electrical & Computer Engineering
University of California, Irvine

Abstract

Distributed object computing (DOC) middleware shields developers from many tedious and error-prone aspects of programming distributed applications. Without proper support from the middleware, however, it can be hard to evolve distributed applications after they are deployed. Therefore, DOC middleware should support meta-programming mechanisms, such as smart proxies and interceptors, that improve the adaptability of distributed applications by allowing their behavior to be modified without changing existing software drastically.

This paper presents three contributions to the study of meta-programming mechanisms for DOC middleware. First, it illustrates, compares, and contrasts several meta-programming mechanisms from an application developer's perspective. Second, it outlines the key design and implementation challenges associated with developing smart proxies and portable interceptors features for CORBA. Third, it presents empirical results that pinpoint the performance impact of smart proxies and interceptors. Our goal is to help researchers and developers determine which meta-programming mechanisms best suit their application requirements.

1 Introduction

Motivation: Developers of distributed applications face many challenges stemming from inherent and accidental complexities, such as latency, partial failure, and non-portable low-level OS APIs. The magnitude of these complexities–combined with increasing time-to-market pressures–make it increasingly impractical to develop distributed applications manually from scratch. Commercial-off-the-shelf(COTS) distributed object computing (DOC) middleware helps address these challenges by:

1. Defining standard higher-level programming abstractions, such as distributed object interfaces, that provide location transparency to clients and server components;

2. Shielding application developers from low-level concurrent network programming details, such as connection management, data transfer, parameter (de)marshaling, endpoint and request demultiplexing, error handling, multithreading, and synchronization; and

3. Amortizing software lifecycle costs by leveraging previous development expertise and capturing implementations of key patterns in reusable middleware frameworks and common services.

In the case of standards-based DOC middleware, such as CORBA [1], these capabilities are realized via an open specification process. The resulting products can interoperate across many OS/network platforms and programming languages [2].

To date, CORBA middleware has been used successfully to enable developers to create applications rapidly that can meet a particular set of requirements with a reasonable amount of effort. CORBA has been less successful, however, at shielding developers from the effects of requirement or environmental changes that occur late in an application's life-cycle, *i.e.*, during deployment and/or at run-time. To address this problem, this paper describes and evaluates *meta-programming mechanisms*, which improve the adaptability of distributed applications by allowing their behavior to be modified with little or not change to existing application software.

The two meta-programming mechanisms we focus on in this paper are:

- **Smart proxies,** which are application-provided stub implementations that transparently override the default stubs created by an ORB to customize client behavior on a per-interface basis.

- **Interceptors,** which are objects that an ORB invokes in the path of an operation invocation to monitor or modify the behavior of the invocation transparently.

These two meta-programming mechanisms can be used to configure new or enhanced functionality into CORBA applications with minimal impact on existing software. The material presented in this paper is based on our experience implementing and using smart proxies and interceptors in TAO [3], which is a open-source, CORBA-complaint ORB designed to support applications with demanding quality-of-service (QoS) requirements.

Paper organization: The remainder of this paper is structured as follows: Section 2 presents an overview of the *smart proxy* and *interceptor* meta-programming mechanisms; Section 3 describe the patterns that guided the development of TAO's smart proxy and interceptor mechanisms and resolved key design challenges; Section 4 illustrates the performance characteristics of TAO's smart proxy and interceptor mechanisms; Section 5 compares our work with related research; and Section 6 presents concluding remarks.

2 Overview of Smart Proxies and Interceptors

DOC middleware provides *stub* and *skeleton* mechanisms that serve as a "glue" between the client and servants, respectively, and the ORB. For example, CORBA stubs implement the *Proxy* pattern [4] and marshal operation information and data type parameters into a standardized request format. Likewise, CORBA skeletons implement the *Adapter* pattern [4] and demarshal the operation information and typed parameters stored in the standardized request format.

CORBA stubs and skeletons can be generated automatically from schemas defined using the OMG Interface Definition Language (IDL). A CORBA IDL compiler transforms application-supplied OMG IDL definitions into stubs and skeletons written using a particular programming language, such as C++ or Java. In addition to providing programming language and platform transparency, an IDL compiler eliminates common sources of network programming errors and provides opportunities for automated compiler optimizations [5].

Traditionally, the stubs and skeletons generated by an IDL compiler are *fixed*, *i.e.*, the code emitted by the IDL compiler is determined at translation time. This design shields application developers from the tedious and error-prone network programming details needed to transmit client operation invocations to server object implementations. Fixed stubs and skeletons make it hard, however, for applications to adapt readily to certain types of changes in requirement or environmental conditions, such as:

- The need to monitor system resource utilization may not be recognized until after an application has been deployed.
- Certain remote operations may require additional parameters in order to execute securely in a particular environment.
- The priority at which clients invoke or servers handle a request may vary according to environmental conditions, such as the amount of CPU or network bandwidth available at run-time.

In applications based on CORBA middleware with conventional fixed stubs/skeletons, these types of changes often require re-engineering and re-structuring of existing application software. One way to minimize the impact of these changes is to devise *meta-programming mechanisms* that allow applications to adapt to various types of changes with little or no modifications to existing software. For example, stubs, skeletons, and certain points in the end-to-end operation invocation path can be treated as *meta-objects* [6], which are objects that refine the capability of base-level objects, which are the objects comprising the bulk of application programs.

As shown in Figure 1, CORBA ORBs are responsible for transmitting client operation invocations to target objects. When a client invokes an operation, a stub implemented as

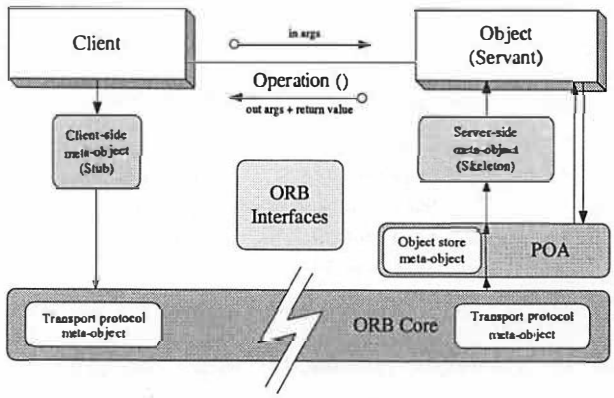


Figure 1: Interactions Between Requests and Meta-objects End-to-End

a meta-object can act in conjunction with transport-protocol meta-objects to access and/or transform a client operation invocation into a message and transmit it to a server. Corresponding meta-objects on the server's request processing path can access and/or perform inverse transformations on the operation invocation message and dispatch the message to its servant. An invocation result is delivered in a similar fashion in the reverse direction.

As all operation invocations pass through meta-objects, certain aspects of application and middleware behavior can be adapted transparently when system requirements and environmental conditions change by simply modifying the meta-objects. To modify meta-objects, the DOC middleware can either (1) provide mechanisms for developers to installed customized meta-objects for the client or (2) embed *hooks* implementing a *meta-object protocol* (MOP)[6] in the meta-objects and provide mechanisms to install objects implementing the MOP to strategize these meta-object behaviors. In the context of CORBA, *smart proxies* are customized meta-objects and *interceptors* are objects that implement the MOP.

2.1 Overview of Smart Proxies

Most CORBA application developers use the fixed stubs generated by an IDL compiler without concern for how the stubs are implemented. There are situations, however, where the default stub behavior is inadequate. For example, an application developer may wish to change stub code transparently in order to:

- Perform application-specific functionality, such as logging;
- Add parameters to a request;
- Cache requests or replies to enable batch transfer or minimize calls to a remote target object, respectively;
- Support advanced quality-of-service (QoS) features, such as load balancing and fault-tolerance; or
- Enforce security mechanisms, such as authentication of credentials.

To support these capabilities *without* modifying existing client code, applications must be able to override the default stub implementations selectively. These application-defined stubs are called *smart proxies*, which are customizable meta-objects that can mediate access to target objects more flexibly than the default stubs generated by an IDL compiler. Smart proxies allow developers to modify the behavior of interfaces without re-implementing client applications or target objects.

The two main entities in smart proxy designs are (1) the smart proxy factory and (2) the smart proxy meta-object, which are shown in Figure 2. When using a smart proxy

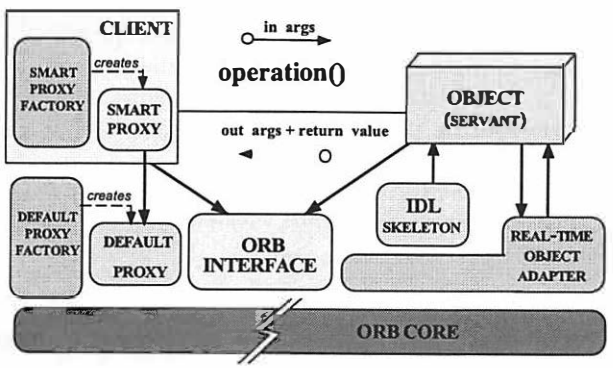


Figure 2: TAO's Smart Proxy Model

to modify the behavior of an interface, the developer implements the smart proxy class and registers it with the ORB. After installing the smart proxy factory, the ORB automatically uses the application-supplied factory to create object references when a client invokes the `_narrow` operation of an interface. Thus, if smart proxies are installed before a client accesses these interfaces, the client application can transparently use the new behavior of the proxy returned by the factory.

Smart proxies are not yet standardized in CORBA, though many ORBs support this feature as a proprietary extension.

2.2 Overview of Interceptors

The smart proxies feature outlined above is a meta-programming mechanism that increases the flexibility of *client* applications. *Interceptors* are another meta-programming mechanism used in DOC middleware to increase the flexibility of both client *and* server applications. In CORBA, interceptors are standard meta-objects that stubs, skeletons, and certain points in the end-to-end operation invocation path can invoke at predefined "interception points."

Prior to CORBA 2.3.1 interceptors were under-specified and therefore non-portable. In contrast, the interceptors discussed in this paper are based on the so-called "Portable Interceptors" specification [7], which is being ratified by the OMG. Two types of interceptors are defined in the CORBA Portable Interceptor specification:

- *Request interceptors*, which deal with operation invocations;
- *IOR interceptors*, which insert information into interoperable object references (IORs).

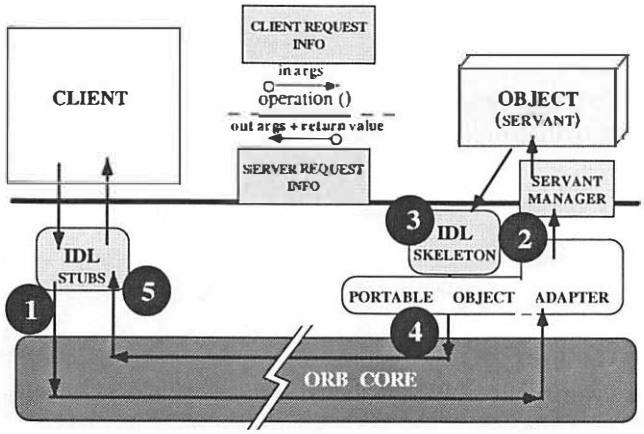
Both types of interceptor are described below.

2.2.1 Request Interceptors

Request interceptors can be decomposed into *client request* interceptors and *server request* interceptors, which are designed to intercept the flow of a request/reply sequence through the ORB at specific points on clients and servers, respectively. Developers can install instances of these interceptors into an ORB via an IDL interface defined by the Portable Interceptor specification. Regardless of what interface or operation is invoked, after request interceptors are installed they will be called on *every* operation invocation at the pre-determined ORB interception points shown in Figure 3.

As shown in this figure, request interception points occur in multiple parts of the end-to-end invocation path when a client sends a request, when a server receives a request, when a server sends a reply, and when a client receives a reply. Different hook methods will be called at different points in this interceptor chain. For example, the `send_request` hook is called on the client before the request is marshaled and the `receive_request` hook is called on the server after the request is demarshaled.

Compared to a client invocation path, a server invocation path has an additional interception point called




```
PORTABLE INTERCEPTOR API :
1) send_request()/send_poll()
2) receive_request_service_contexts ()
3) receive_request()/receive_poll()
4) send_reply()/send_exception()/send_other()
5) receive_reply()/receive_exception()/
   receive_other()
```

Figure 3: Request Interception Points in the CORBA Portable Interceptor Specification

`receive_request_service_contexts`, which is invoked before the POA dispatches a servant manager. This interception point prevents unnecessary upcalls to a servant. For example, in the CORBA Security Service [8] framework this interception point can be used to inspect security-related credentials piggybacked in a service context list entry. If the credentials are valid the upcall can proceed to other interceptors (if they exist) or to the servant; if not, an exception will be returned to the client.

The behavior of an interceptor can be defined by an application developer. An interceptor can examine the state of the request that it is associated with and perform various actions based on the state. For example, interceptors can invoke other CORBA operations, access information in a request, insert/extract piggybacked messages in a request's service context list, redirect requests to other target objects, and/or throw exceptions based on the object the original request is invoked upon and the type of the operation. Each of these capabilities is described below:

Nested invocations: A request interceptor can invoke operations on other CORBA objects before the current invocation it is intercepting completes. For example, monitoring and debugging utilities can use this feature to log information associated with each operation invocation. To avoid causing infinite recursion, developers must be careful to act only on targeting interfaces and operations they intend to affect when performing nested invocations in an interceptor.

Accessing request information: Request interceptors can access various information associated with an invocation, such as the operation name, parameters, exception lists, return values, and the request id via the MOP interface as defined in the Portable Interceptor specification. Interceptors cannot, however, modify parameters or return values. This request/reply information is encapsulated in an instance of `ClientRequestInfo` or `ServerRequestInfo` classes, which derive from the `RequestInfo` class and contain the information listed above for each invocation.

For example, client request interceptors are passed `ClientRequestInfo` and server request interceptors are passed `ServerRequestInfo`. These `RequestInfo`-derived objects can use features provided by the CORBA `Dynamic` module. This module is a combination of pseudo-IDL types, such as `RequestContext` and `Parameter`, declared in earlier CORBA specifications. These types facilitate on-demand access of request information from the `RequestInfo` to avoid unnecessary overhead if an interceptor does not need all the information available with the `RequestInfo`.

Service context manipulation: As mentioned earlier, request interceptors cannot change parameters or the return value of an operation. They can, however, manipulate *service contexts* that are piggybacked in operation requests and replies exchanged between the clients and servers. A service context is a sequence field in a GIOP message that can transmit "out-of-band" information, such as authentication credentials, transaction contexts, operation priorities, or policies associated with requests.

For example, the CORBA Security Service uses request interceptors to insert user identity via service contexts. Likewise, the CORBA Transaction Service uses request interceptors to insert transaction-related information into service contexts so it can perform extra operations, such as commit/rollback, based on the operation results in a transaction. Each service context entry has a unique service context identifier that applications and CORBA components can use to extract the appropriate service context.

Location forwarding: Request interceptors can be used to forward a request to a different location, which may or may not be known to the ORB *a priori*. This can be done via the `PortableInterceptor::ForwardRequest` exception, which allows an interceptor to inform the ORB that a retry should occur upon the new object indicated in the exception. The exception can also indicate whether the new object should be used for all future invocations or just for the forwarded request.

Since the `ForwardRequest` exception can be raised at most interception points, it can be used to provide fault tol-

erance and load balancing [9]. For example, the IOR of a replicated object can be used as the forward object in this exception. When the object dies for some reason–and this situation is conveyed to the interceptor–this exception can be raised even before the POA tries to make an upcall.

Multiple interceptors: Multiple request interceptors can be registered with an ORB, which will then iterate through them and invoke the appropriate interception operation at every interception point according to the following rules:

- For each request interceptor, only one *starting* interception point can be called for a given invocation. A starting interception point is the first point invoked in a request/reply sequence. For instance, the starting points for a client ORB include `send_request` and `send_poll`. Likewise, the starting point for a server ORB is `receive_request_service_contexts`.
- For each request interceptor, only one *ending* interception point can be called for a given invocation. The ending interception point is the last juncture where an interception may occur in the request/reply sequence. The ending interception points on a client ORB are `receive_reply`, `receive_exception`, and `receive_other` and the ending interception points for a server ORB consist of `send_reply`, `send_exception`, and `send_other`.
- There can be multiple intermediate interception points.
- Intermediate interception points cannot be invoked in the case of an exception.
- The ending interception point for a given interceptor will be called only if the starting interception point runs to completion.

Multiple interceptors are invoked using a flow-stack model. When initiating an operation invocation, an interceptor is pushed onto the stack after its starting interception point completes successfully. When an invocation completes, the interceptors are popped off the stack and invoked in reverse order. The flow-stack model ensures that only interceptors executed successfully for an operation can process the reply/exceptions.

Exception handling: Request interceptors can affect the outcome of a request by raising exceptions in the inbound or outbound invocation path. In such cases, the `send_exception` operation of a server request interceptor is invoked on the reply path and is received at the client in the `receive_exception` interceptor hook. When a `send_exception` or `receive_exception` operation raises a `ForwardRequest` exception, the other interceptors have their `send_other` and `receive_other` interception points invoked, respectively.

2.2.2 IOR Interceptors

IIOP version 1.1 introduced an attribute called `components`, which contains a list of *tagged components* to be embedded within an IOR. When an IOR is created, tagged components provide a placeholder for an ORB to store extra information pertinent to the object. This information can contain various types of QoS information related to security, server thread priorities, network connections, CORBA policies, or other domain-specific information.

The original IIOP 1.0 specification provided no standard way for applications or services to add new tagged components into an IOR. Services that require this field were therefore forced to use proprietary ORB interfaces, which impeded their portability. The Portable Interceptors specification resolves this problem by defining *IOR interceptors*.

IOR interceptors are objects invoked by the ORB when it creates IORs. They allow an IOR to be customized, *e.g.*, by appending tagged components. Whereas request interceptors access operation-related information via `RequestInfos`, IOR interceptors access IOR-related information via `IORInfos`. Figure 4 illustrates the behavior of IOR interceptors. A server

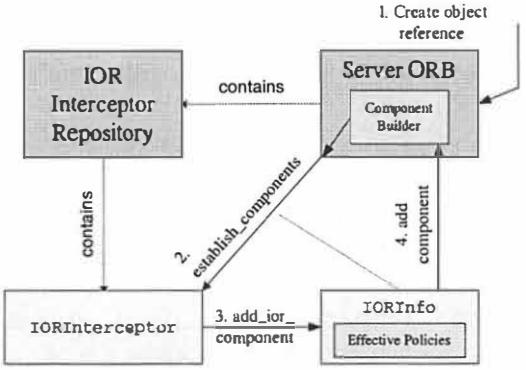


Figure 4: IOR Interceptors

ORB responsible for creating an IOR contains an *IOR interceptor repository*. In turn, this repository contains a series of IOR interceptors that have been registered with the ORB. When the server process requests the ORB to create an IOR, the ORB iterates through the IOR interceptors in the repository using the `establish_components` operation. The IOR interceptors then add tagged components to the IOR being generated by refering to the `IORInfo` passed in by calling `add_ior_component` or `add_ior_component_to_profile`.

2.3 Evaluating Alternative Meta-Programming Mechanisms for ORB Middleware

We presented an overview of smart proxies and interceptors above. We now evaluate these two mechanisms, and then

compare and contrast them with two other meta-programming mechanisms–pluggable protocols and servant managers–that are provided by most CORBA implementations.

2.3.1 Smart Proxies vs. Interceptors

Smart proxies and interceptors are similar in that they extend ORB-mediated invocations and functions. They differ, however, in their architecture and have their own pros and cons, as described below.

Intent: A smart proxy can be used for a variety of purposes, such as improving performance via caching, whereas interceptors are used primarily to (1) audit and verify information along the invocation path and (2) redirect the operation if necessary. For instance, a server request interceptor can determine whether the server should handle certain operation invocations by inspecting the incoming requests and forwarding some requests to other servers that can handle them.

Scope of control: A different smart proxy can be configured for each interface, whereas the same set of interceptors will be invoked at *all* the ORB mediated points of an invocation. Moreover, a smart proxy is solely a client mechanism, whereas request interceptors are invoked on the request path from client-to-server and on the reply path from server-to-client.[1]

Invocation points: A smart proxy invocation point occurs whenever an operation is invoked through a stub. In contrast, interceptors are invoked at many points, including IOR creation time and/or before a call is sent by the POA to the servant.

Cardinality: A client can have only a single smart proxy for each interface, whereas multiple interceptors can be registered with the ORB.

Modifiability: Since smart proxies replace default ORB generated stubs completely, smart proxies can modify the parameters or results of an operation. In contrast, the Portable Interceptor specification does not allow request interceptors to change operation parameters or return values.

Overhead: A smart proxy mechanism incurs minimal overhead, *i.e.*, a single extra method call per-operation invocation. In contrast, request interceptors can incur additional overhead to access request information because information related to the request is bundled into `any`s, which have higher overhead for their insertion and extraction operations.

Standardization: Smart proxies have not yet been standardized in the CORBA specification. CORBA interceptors will be standardized after the Portable Interceptor specification is ratified.

[1] IOR interceptors are just invoked during object reference creation.

In general, design problems that require pre-invocation or per-interface extensions are well-suited for smart proxies. Portable interceptors, in contrast, provide a suitable solution for applications that require a semantically richer–albeit somewhat more expensive–meta-programming abstraction.

2.3.2 Servant Managers

The CORBA POA specification [1] allows server applications to register *servant manager* objects that activate servants on-demand, rather than creating all servants before listening for requests. There are two types of servant managers in CORBA:

- **Servant activators,** which provide a hook method called `incarnate` that creates a servant the first time an object is accessed by a client.
- **Servant locators,** which provide a hook method called `preinvoke` that are invoked by a POA to create a servant for every request on an object. Figure 5 illustrates how servant locators are used in a CORBA application to perform various resource management activities before dispatching an operation to a servant.

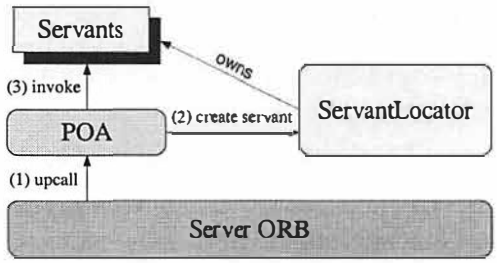


Figure 5: Managing Resources with a Servant Locator

A servant locator is similar to an interceptor in several respects. For example, both are implementations of the Interceptor pattern [10]. Moreover, both can (1) intercept requests before they are dispatched to servants, (2) invoke extra operations, and (3) affect the outcome of request invocations, *e.g.*, by throwing exceptions. Unlike interceptors, however, servant locators only affect the POAs that install them and can only provide access to a limited subset of the request-related information. As a consequence, they are more tightly coupled with POAs and servant implementations than are interceptors.

2.3.3 Pluggable Protocols Frameworks

Another type of meta-programming mechanisms provided by some DOC middleware is *pluggable protocols frameworks* [11, 12], which is in the process of being standardized by the OMG in the Extensible Transport Framework [13] specification effort. These frameworks decouple the ORB's transport protocols from its component architecture. Developers

can therefore add new protocols without requiring changes to existing application software.

Figure 6 illustrates TAO's pluggable protocols framework, which allows developers to install new protocols into the ORB by implementing customized pluggable protocol objects. Higher-level application components and CORBA services

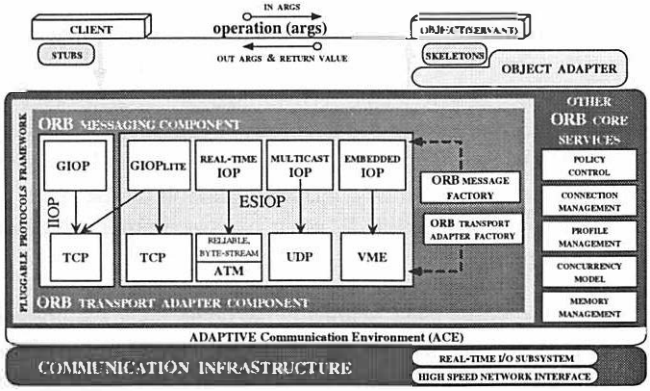


Figure 6: TAO's Pluggable Protocols Framework Architecture

use the Component Configurator pattern [10] to dynamically configure custom protocols into TAO's pluggable protocols framework *without* requiring obtrusive changes to themselves or the ORB.

As with interceptors and smart proxies, pluggable protocols frameworks are meta-programming mechanisms that add functionality to ORBs. However, whereas other two mechanisms alter the semantic of objects, pluggable protocols frameworks alter the underlying ORB transport mechanism. Thus, they do not permit fine-grained control over objects since they affect *all* objects in an ORB and it is hard to vary the transport mechanism at the level of object references. Moreover, since pluggable protocols deal directly with the communication infrastructure, they are usually more complex to program than interceptors or smart proxies.

Figure 7 compares the various meta-programming mechanisms along a number of dimensions described above. Portable interceptors have the highest overhead since they are the most flexible meta-programming mechanism. Although other mechanisms have less overhead compared to portable interceptors, they are targeted at more specific system mechanisms. When combined with patterns, such as Component Configurator [10] and OS features, such as explicit dynamic linking [14], these meta-programming mechanisms can all be configured dynamically into CORBA clients and servers.

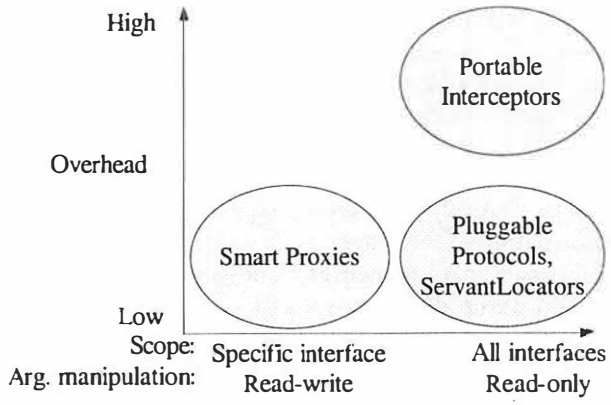


Figure 7: Comparing Alternative Meta-programming Mechanisms

3 Key Design Challenges and Pattern-based Resolutions

In this section, we explore how smart proxies and interceptors are implemented in TAO. To clarify and generalize our approach, the discussion below focuses on the patterns [4] we applied to resolve the key design challenges faced during our development process.

3.1 Smart Proxy Design Challenges and Resolutions

As mentioned in Section 2.1, the goal of using smart proxies is to change/add behaviors to existing programs with minimal modifications to client applications. Below, we discuss the key design challenges we faced while refactoring TAO's existing stub architecture to support smart proxies.

3.1.1 Challenge: Providing Flexible Support for Smart Proxies

Context: The proxy framework generated by TAO's IDL compiler should allow applications to use customized proxies transparently. For example, changes to client applications that use customized proxies must be localized. In particular, developers should be able to install customized proxies with little or no change to client application code.

Problem: TAO's original IDL compiler generated only fixed default proxies. In particular, the `_narrow` operation it generated for each interface returned a default proxy. If developers require more flexibility, however, the `_narrow` operation must be able to return either an IDL-generated default proxy or a custom smart proxy.

Since the `_narrow` operation is generated by TAO's IDL compiler as part of the client's stub it is not possible to mod-

ify this method externally from a client application. Moreover, since fixed default stubs were generated any changes required manually modifying the IDL-generated code. Clearly, this solution was inflexible and had to be solved at the stub-generation level.

Solution → Apply the Factory Method, Adapter, and Singleton patterns: We applied these design patterns [4] in TAO's smart proxy framework to provide the necessary flexibility to create different types of proxies transparently in TAO's IDL-generated code, as follows:

- The Factory Method pattern defers instantiation of various types of meta-objects to subclasses.
- The Adapter pattern provides a higher level of abstraction for TAO's proxy factories and to delegate creation requests to the appropriate factory.
- The Singleton pattern makes the proxy factory adapter a global access point for factory registration from program initialization to termination.

Figure 8 illustrates how we applied these three patterns in TAO to provide flexible support for smart proxies. By using

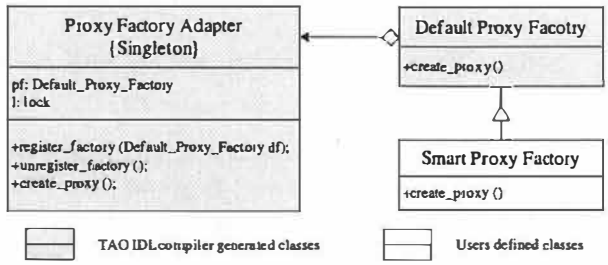


Figure 8: Applying Patterns to Provide Flexible Support for Smart Proxies

these patterns, applications can obtain either the default IDL-generated proxy or a smart proxy without changing existing code manually. For example, after an application registers a per-interface smart proxy factory, the `_narrow` operation call will create the appropriate proxy automatically.

3.1.2 Challenge: Treating Remote and Collocated Smart Proxies Uniformly

Context: A target object can be either remote or it can be collocated in the client's address space [15]. TAO provides customized meta-objects called *collocated proxies* to optimize performance for collocated objects. Smart proxies should provide similar functionality to collocated and remote proxies since the ability to differentiate remote and collocated smart proxies provides developers with greater flexibility.

Problem: Depending on where a target object resides, a developer may or may not wish to invoke the smart proxy installed for the object. For example, a developer may not want to cache operation results in a collocated smart proxy because these calls are already resolved locally. Originally, TAO treated the generation of collocated stubs as a special case and if smart proxies were installed they would supercede the default stubs, even if the stubs were collocated.

Ignoring collocation optimizations, however, may cause unnecessary waste by trying to optimize a bottleneck that does not exist. Therefore, it is necessary to distinguish the remote and collocated case to take full advantage of this construct and avoid unnecessary waste of system resources, such as memory and CPU cycles. In addition, smart proxies must (1) provide applications with the same interface as default proxies and (2) be able to call down to the default proxy to communicate with remote target objects.

Solution → Apply the Composite pattern: The Composite pattern [4] supports part/whole relationships and allows all objects in such composite structures to be processed uniformly. We applied the Composite pattern to TAO to provide a uniform view among different proxies available to clients. As shown in Figure 9, in this design (1) smart proxy classes inherit from the

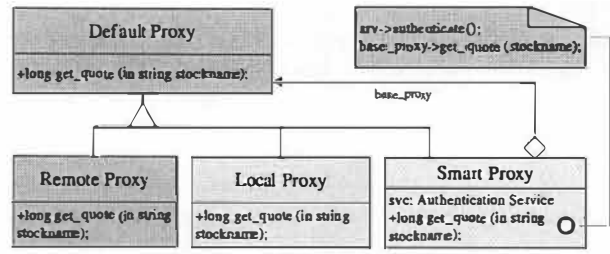


Figure 9: Applying the Composite Pattern to TAO's Smart Proxy Design

default proxy and (2) also store a pointer to the default proxy to make invocations to target object. Collocated and remote proxies are children of the default proxy. Thus, smart proxies can make calls to the remote or collocated proxy transparently, while providing the same application interface as the default proxies.

3.2 Interceptor Design Challenges and Resolutions

As discussed in Section 2.2, interceptors can extend the behavior of CORBA operations with minimal changes to client and server applications. In this section, we discuss the key design challenges faced while enhancing TAO's existing invocation architecture to support interceptors.

3.2.1 Challenge: Making Information Retrieval Possible Per-Operation

Context: Request interceptor hook methods are invoked at different interception points along the invocation path. These interceptors must be able to (1) verify and audit information being passed to the target object as the invocation continues and (2) potentially terminate the invocation before it reaches the target object.

Problem: An ORB must provide information in response to interceptor queries. This information may be operation-specific and even temporal. For example, the result of an operation may be available only after the POA makes an upcall to a servant and the operation executes.

An ORB must therefore have a generic way to access operation-level information and disclose this information to interceptors that are invoked at ORB-mediated interception points. Originally, TAO did not maintain this information to avoid degrading the normal execution of the invocation in situations where this information was not required by applications. However, TAO's original design made it hard for applications to influence invocation behavior.

Solution → Generation of nested RequestInfo classes for each interface operation: To provide invocation information dynamically and efficiently, we modified TAO's IDL compiler to generate `RequestInfo` classes for each operation. `RequestInfo` classes are instantiated for each operation invocation and passed to the interceptors during the invocation. Thus, interceptors can access operation-related information, as shown in Figure 10. Every operation in an IDL interface

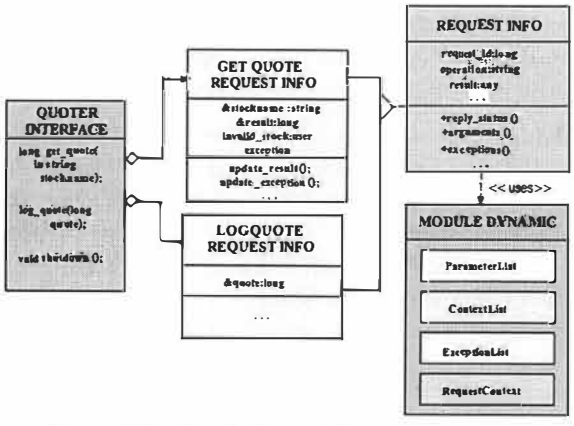


Figure 10: TAO's Portable Interceptor Design

may have different formal parameters, result types, and user exceptions. To minimize the overhead of copying multiple arguments and the return value of the upcall, we only store a reference, rather than a copy of the parameters, results, and exceptions.

We added TAO-specific methods to each `RequestInfo` class and used these methods internally to update the result and the exception thrown, rather than instantiating a new `RequestInfo` class before every interception point is called. For instance, the result of an operation is obtained only after the POA makes the upcall and the client receives a reply. At this point, the client can verify the result in the `receive_reply` interceptor hook by querying the `RequestInfo` object, making it necessary to update the result before this interception point is invoked. Thus, temporal information can also be propagated to interceptors.

3.2.2 Challenge: Avoiding Gratuitous Waste Constructing RequestInfos

Context: Interceptors can access any request-related information. Their interface must therefore be sufficiently general to incorporate any type of data. In CORBA, `any` is a generic type that can hold information of any other types, which are stored using type/value tuples.

Problem: In general, not all interceptors installed in an ORB are interested in handling all information, or even all operations. For example, security-related interceptors may not be interested in what operation is being invoked, but only want to know the contents of the service context list. Likewise, an auditing interceptor may only be interested in the parameters of certain operations of certain objects, while ignoring others altogether.

Although CORBA's `any` type is flexible, it is less efficient and more resource consumptive than other common CORBA data types, such as `long` or `struct`. We need to avoid the overhead of `any` insertion operators if installed interceptors are not interested in certain operation information. There is no way, however, to predict what interceptors will be interested in *a priori*.

Solution → On-demand creation of operation information: To avoid unnecessary waste of resources, we applied the Lazy Initialization pattern [16] to make sure the operation information is only inserted into `any` objects the *first time* a related interface is accessed by an interceptor via its `RequestInfo`-derived interface. This design ensures that pertinent information in `RequestInfo`-derived objects will only be created if an interceptor is interested in the information. In TAO, we retrieve this information via types defined in the CORBA `Dynamic` module.

The `Dynamic` module defines the collocation of request parameters, results, and exceptions in `any` in a sequence of structures that an application interceptor can extract and use. In TAO, methods returning Dynamic objects are implemented to minimize the gratuitous waste of storing all information de `facto` into lists of `any`s as shown in Figure 11. In particu-

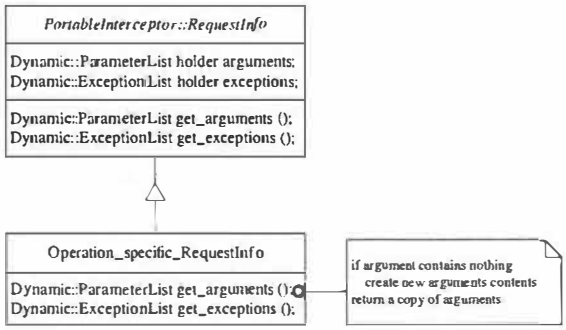


Figure 11: TAO applies Lazy Initialization building Dynamic objects in RequestInfo

lar, this information is inserted into anys only when queried, which occurs just once. Subsequent queries simply return the any variables created previously. Thus, unless an interceptor needs to query a particular piece of request information, it incurs no additional overhead. This optimization is targeted for the common case where interceptors are used to pass service contexts.

3.2.3 Challenge: Implementing Time and Space Efficient Flow Stacks

Context: The Portable Interceptor specification defines *general flow rules* to which a portable interceptor implementation should adhere. These rules ensure that only interceptors invoked successfully from a starting interception point will ever be invoked at an ending interception point. Conceptually, interceptors are pushed on to a stack if invoked successfully in a starting interception point and popped off that stack when they invoked at ending interception points.

Problem: To implement the semantics dictated by CORBA's general flow rules, some type of stack implementation is needed. However, implementing a *flow* stack with a general-purpose stack container class, such as the one in the standard C++ library [17], has the following problems:

- **Time overhead:** The stack implementation may incur non-trivial performance overhead if it allocates space off of the heap dynamically for each interceptor or interceptor reference pushed onto the stack. Dynamic memory is particularly problematic for real-time applications.

- **Space overhead:** The stack implementation itself adds to the ORB footprint since a template must be instantiated for each type of request interceptor, *i.e.*, client or server request interceptors. Moreover, other auxiliary templates may need to be instantiated for internal stack support code. Not only does this increase the static footprint of the ORB, but it also increases run-time ORB memory requirements, which may be unacceptable for embedded applications.

In addition to inherent problems with real stack implementations detailed above, another common problem can occur. Since interceptors are invoked during a request, they are in the critical path. This means that interceptor support code, such as a flow stack, can have an adverse affect on performance if that support is not implemented efficiently. In particular, adding locking mechanisms in the flow of a request can degrade performance since threads waiting for a lock can block. The act of acquiring and releasing the lock also imposes further delays.

Solution → Apply optimization principle patterns: Optimization principle patterns [18] define a set of principles that can be applied to improve performance in various ways. To implement time and space efficient flow stacks, heap allocations must be minimized to avoid degrading performance and increasing footprint. Both can be avoided by taking advantage of *pre-computed* resources and the properties associated with them.

As dictated by the Portable Interceptor specification, interceptors are registered with the ORB when the ORB is bootstrapped, *i.e.*, during the initial `CORBA::ORB_init` call. This means that storage for the interceptors will already have been allocated by the time the interceptors are invoked so there should ideally be no need for additional allocations at a later point in time.

By keeping the order with which the interceptors are stored unchanged for the lifetime of the ORB, it is possible to implement highly efficient stack *push* and *pop* operations. Interceptors will always be pushed on to the stack with the same relative ordering they are stored in the ORB. This property ensures that the number of elements on the stack will be equal to the ORB storage location of the last interceptor pushed on to the stack. Hence, the general flow rule semantics can be implemented using a *logical* flow stack.

Applying the solution to TAO: TAO stores pointers to registered interceptors in a pre-allocated array, which avoids increased footprint and run-time memory requirements. Rather than having to instantiate a stack for each type of interceptor (*i.e.*, client and server request interceptors), a single array for each type of request interceptor is created. The order in which interceptors are stored in the array remains unchanged for the lifetime of the ORB. Thus, *push* and *pop* operations can be implemented by simply incrementing and decrementing a variable, respectively, as illustrated in Figure 12.

The following example presents a scenario that illustrates how TAO's logical flow stacks are implemented:

1. Three request interceptors are registered when the ORB is initialized. Specifically, the `CORBA::ORB_init` method invokes all ORB initializers registered by the application. Those ORB initializers then register the interceptors by using the appropriate methods in the `ORBInitInfo` argument passed to the ORB initializer by the `CORBA::ORB_init`

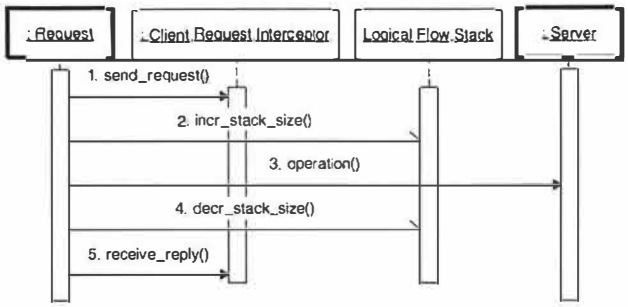


Figure 12: An Efficient Flow Stack Implementation

method. An example of this interceptor registration code follows:

```
// Code that would reside in a
// concrete implementation of an
// ORBInitializer::post_init() method, for
// example.

// Create and install a client interceptor.
PortableInterceptor::
ClientRequestInterceptor_var
  interceptor = new
    Secure_Client_Request_Interceptor;

// "info" is the ORBInitInfo argument.
info->add_client_request_interceptor
  (interceptor.in ());
```

2. Two interceptors are successfully invoked at a *starting* interception point during a request. This corresponds to step 1 in Figure 12.

3. Each successful request interceptor invocation increments the stack size by one, which results in a stack size of two. Stack element one corresponds to request interceptor one as stored in the ORB's interceptor array. Similarly, stack element two corresponds to interceptor two in the ORB's interceptor array. Again, a *logical* stack is in use here. This corresponds to step 2 in Figure 12.

4. An *ending* interception point is invoked.

5. Within the ending interception point, each of the interceptors in the logical stack is invoked. Prior to invoking each interceptor, the stack size is decreased by one (step 4 in Figure 12), effectively popping an interceptor off of the logical flow stack. Since only the first two interceptors were pushed on to the stack, only the first two of the three interceptors will be invoked (step 5 in Figure 12) in the ending interception point and the third interceptor will never be invoked.

TAO's logical flow stack implementation allows the CORBA general flow rule semantics to be implemented efficiently and with minimal impact on ORB footprint. These benefits arise from the fact that flow stack storage is preallocated prior to the first use of the flow stack. In addition, the TAO implementation ensures the order of the interceptors stored in the ORB's interceptor array remains unchanged for the lifetime of the ORB.

One other aspect of this implementation is the fact that it is *not* necessary to acquire a lock to prevent other threads from modifying the logical stack. Only one thread ever services a request at a given time. Thus, there is no need to implement a locking mechanism for the logical stack, in which case additional overhead is not incurred.

4 Empirical Benchmarking Results

Developers of distributed applications must often make tradeoffs between time/space overhead and flexibility. Selecting which meta-programming mechanism to use, *e.g.*, smart proxies or interceptors, is an example of this tradeoff. This section presents benchmarking results that quantify the time/space overhead and tradeoffs of using smart proxies and portable interceptors.

4.1 Overview of the Testbed Environment and Benchmarks

The experiments were conducted using a Bay Networks LattisCell 10114 ATM switch connected to two dual-processor UltraSPARC-2s running SunOS 5.7. Each UltraSPARC-2 contains 2 168 MHz CPUs with a 1 Megabyte cache per-CPU, 256 Mbytes of RAM, and an ENI-155s-MF ATM adapter card that supports 155 Megabits per-sec (Mbps) SONET multimode fiber. The experimental testbed is shown in Figure 13. The benchmarking programs were compiled using the Sun CC

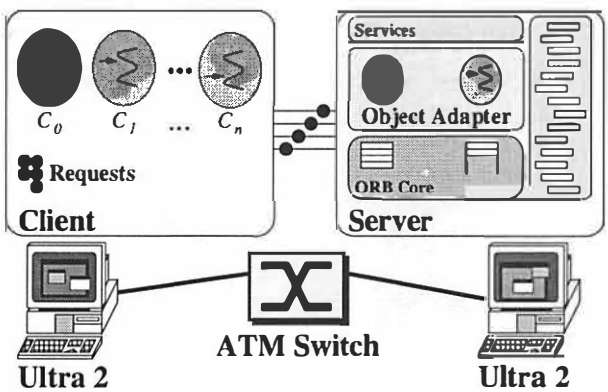


Figure 13: Testbed for Meta-programming Mechanism Benchmarks

5.0 compiler with all optimizations enabled. We conducted

two different benchmarks: one measured the performance of smart proxies and the other the performance of interceptors.

4.1.1 Smart Proxy Results

The overhead of calling an operation via a smart proxy is equivalent to calling the default proxy, *i.e.*, it is the cost of a local virtual method call. Therefore, we designed our smart proxy benchmark to show how performance can be improved if smart proxies are used as a cache to minimize the number of remote operations. Here is the IDL interface we used for this test:

```
interface Broadway_Show
{
  // Get the prices for the box
  // seats of the Broadway show.
  short box_prices ();

  // Order tickets.
  long order_tickets (in short number);
};
```

The servant in the test is a virtual box office that allows clients to purchase tickets to Broadway shows. A client can query the prices of box seats and if they are within a price range, it buys them. Thus, the client normally makes two invocations: (1) `box_prices` and (2) `order_tickets` if the prices are reasonable. By default, every time a client enquires about ticket prices, a remote invocation occurs.

We can minimize overhead significantly by using a smart proxy that makes just one remote invocation and then caches the result and reuses it when subsequent enquiries occur. This caching improves the performance significantly, as shown in Figure 14. This figure illustrates that omitting unnecessary

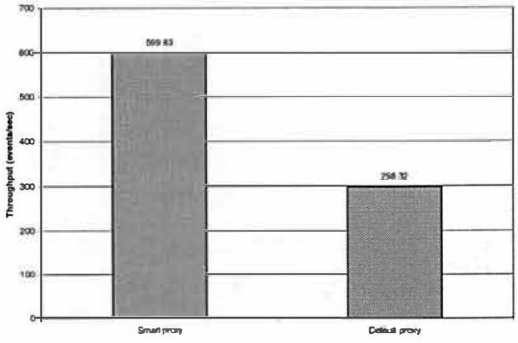


Figure 14: Performance Improvement Using a Smart Proxy to Cache Information

remote operation calls improve the performance by ~130%, even over a high-speed ATM network.

4.1.2 Portable Interceptor Results

Our portable interceptor benchmarks quantify the cost of supporting and using interceptors in TAO. Moreover, these tests quantified the costs of individual interceptor features, such as accessing a parameter list and accessing a service context list. In the benchmark program, the following three IDL operations were defined in the `Secure_Vault` interface:

```
interface Secure_Vault
{
  exception Invalid {};

  struct Record { long check_num; long amount; };

  // No args/exceptions operation.
  short ready ();

  // Throws a user exception.
  void authenticate (in string user)
    raises (Invalid);

  // updates a struct and returns a count.
  long update_records (in long id,
                       in Record val);
};
```

Each operation takes a different number and different length of parameters and return values. Moreover, the `authenticate` operation throws a user exception, whereas the other two do not. This diversity allowed us to measure the cost of preparing different types of generic information required by interceptors.

The interceptor benchmarks were run using the five different configurations summarized below:

1. No interceptor support: In this configuration, interceptor support was disabled completely in the ORB, which measured TAO's baseline performance.

2. No interceptor installed: This time the ORB was compiled with interceptor support, although the test was performed without installing an interceptor into the ORB. This configuration measures the performance penalty applications must pay for the potential of flexibility.

3. No-op interceptor installed: This configuration uses a no-op interceptor to measure the cost of invoking interceptors.

4. Accessing the service context list: The interceptor installed in this configuration manipulates the GIOP request's `ServiceContextList`. On the client, a request interceptor creates a new `ServiceContext` containing an encapsulated password string of 7 bytes and inserts the service context object into the `ServiceContextList` of the invocation using the `RequestInfo` interface. On the server, a different request interceptor performs the reverse operation by (1) extracting the password string from the `ServiceContextList` using the `RequestInfo` interface and (2) examining the password via a string comparison.

5. Accessing Dynamic information: TAO implements the `Dynamic` module types in request/reply operations, such as parameters, results and exception list of an invocation, by creating these information on-demand. The interceptor installed in this configuration accesses the dynamic information of the operations by checking their parameters and return values.

Figure 15 shows the cost of supporting and using these various features and configurations in interceptors. In the first con-

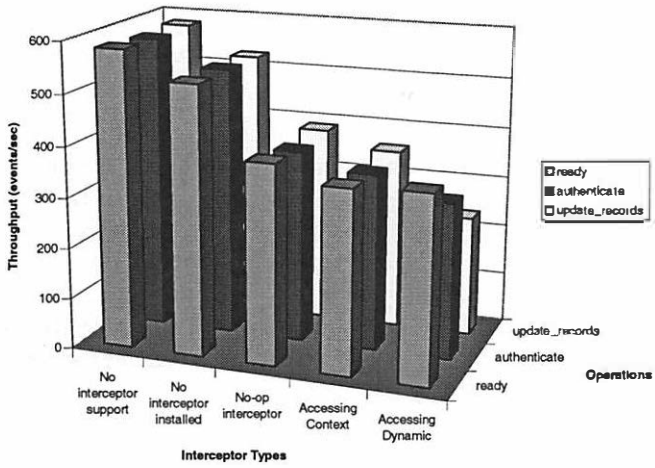


Figure 15: Cost of Using Various Interceptor Features

figuration (no interceptor support), all three measured operations perform similarly because there is no significant difference between the information these operations exchange. The results are similar for the second configuration, which added interceptor support to the ORB but without installing any interceptors. There is only a ~9% performance penalty for using the ORB with interceptor support.

The no-op interceptor provide the baseline cost of invoking an interceptor. There is ~26% of performance penalty compared to not installing the interceptor due to invocations of interception points on every operation invocation. As shown in Figure 15, however, all three operations reveal similar performance characteristics, regardless of the number and size of their parameters and return values.

Similar performance degradation is also observed for interceptors that access the `ServiceContextList`. This configuration measures the cost of adding and extracting a short string from the `ServiceContext`. Again, all three operations experience ~8% degradation in performance compared to using the no-op interceptor.

The interceptor that access the `Dynamic` module types, however, demonstrates more diversity in performance degradation among the three operations we tested. There are ~7%, ~19%, ~and 40% performance hits to the `ready`, `authenticate`, and `update_record` operations, respectively, compared with no-op interceptor configuration. The performance penalty comes not only from the accessing parameters using the `Dynamic` module types, but also from the on-demand creation of the dynamic information. The results show that the preparation of `Dynamic` module types are expensive, which justifies our decision not to create them if they are not accessed by interceptors.

4.2 Memory Footprint Results

TAO is an open-source ORB that is used for real-time and embedded systems with memory constraints. Therefore, smart proxies and interceptors can be conditionally compiled in or out at ORB compile-time. To measure the memory increment necessary to support smart proxies and interceptors, we compiled the `Secure_Vault` IDL interface shown above with three different operations using the following configurations:

1. Interceptors and smart proxies disabled.
2. Interceptors and the smart proxies both enabled;
3. Interceptors enabled but smart proxies disabled, which is the default configuration in TAO; and
4. Interceptors disabled and smart proxies enabled.

Table 1 shows the resulting sizes for different configurations. Not counting the application-specific proxy and factory

Supporting Config.	Stub size (KB)	% Inc.	Skeleton size (KB)	% Inc.
Neither	1,288	0	1,277	0
Smart proxies	1,321	2.5	1,277	0
Interceptors	1,479	14.8	1,485	16.3
Both	1,517	17.8	1,489	16.6

Table 1: Footprint Comparison for Smart Proxies and Interceptors

method, smart proxies increase TAO's client memory footprint by ~2.5%. In contrast, interceptors require ~15% extra footprint to handle on-demand creation of parameters lists, exceptions list, etc.

We also performed the same test using the OMG *Minimum CORBA* configuration [19], which defines a subset of the complete ORB CORBA specification to reduce embedded system memory footprints. By default, TAO's Minimum CORBA footprint is less than 1 MB. To determine the footprint growth when smart proxies and/or interceptors are used, we measured the size of the ORB again using the same IDL interface, as shown in Table 2: The footprint increase for TAO's smart

Supporting Config.	Stub size (KB)	% Inc.	Skeleton size (KB)	% Inc.
Neither	923	0	896	0
Smart proxies	974	5.5	896	0
Interceptors	1,115	20.7	1,104	23.1
Both	1,148	24.3	1,105	23.2

Table 2: Footprint Comparison for Smart Proxies and Interceptors in TAO's Minimum CORBA Configuration

proxies in this configuration is 5.55% and the support for interceptors causes a significant 20-23% increment. These results are not surprising since both these meta-programming features are new and have not yet been optimized for TAO's Minimum CORBA configuration.

In general, the results in this section show that CORBA meta-programming mechanisms can provide developers with significant improvements in functionality, performance, and convenience without drastic changes to existing application software. Depending on which features are used, however, developers need to consider the affect of time and space overhead.

5 Related Work

CORBA is increasingly being adopted as the middleware of choice for a wide-range of distributed applications and systems. As systems evolve, new features/services will be added to the system. Smart proxies and interceptors are good ways to adapt existing applications to take advantage of these new features. The following work on middleware technologies is related to our research.

QuO: The *Quality Objects* (QuO) distributed object middleware is developed at BBN Technologies [20] by applying Aspect-Oriented Programming (AOP) [21] techniques to adaptive network applications. QuO is based on CORBA and supports:

1. *Run-time performance tuning and configuration* through the specification of operating regions, behavior alternatives, and reconfiguration strategies that allows the QuO run-time to adaptively trigger reconfiguration as system conditions change, represented by transitions between operating regions; and

2. *Feedback* across software and distribution boundaries based on a control loop in which client applications and server objects request levels of service and are notified of changes in service.

QuO achieves this functionality via customized smart proxies, called *delegates*, and embedded MOP interfaces within the proxies. However, their framework does not allow users to install user-defined proxies and the MOP interfaces are specifically designed for QoS purpose.

Orbix filters: Orbix defines the concept of filters, which are an interceptor mechanism based on the concept of "flexible bindings" [22]. By deriving from a predefined base class, developers can intercept events. Common events include client-initiated transmission and arrival of remote operations, as well as the object implementation-initiated transmission and arrival of replies. Developers can choose whether to intercept the request or result before or after marshaling. Orbix programmers can leverage the same filtering mechanism to build multi-threaded servers [23, 24, 25].

dynamicTAO: The dynamicTAO reflective ORB [26] supports interceptors for monitoring and security. Particular interceptor implementations are loaded into dynamicTAO using the Component Configurator pattern [10]. Using component configurators to install interceptors in dynamicTAO allows applications to exchange monitoring and security strategies at run-time. Moreover, there are extensive use of reflective programming technique in dynamicTAO to determine the module the ORB requires.

Fault-tolerant ORB frameworks: Interceptors have been applied in a number of fault-tolerant ORB frameworks such as the Eternal system [27]. Eternal intercepts system calls made by clients through the lower-level I/O subsystem and maps these system calls to a reliable multicast subsystem. Eternal does not modify the ORB or the CORBA language mapping, thereby ensuring the transparency of fault tolerance from applications.

COM interceptors: Hunt and Scott [28] describe how to implement interceptors in COM. The concept they use to implement interceptors is similar to TAO's collocated stub [15]. This technique uses alternative wrappers around the object implementation to masquerade as operation targets, which are similar to TAO's smart proxies.

6 Concluding Remarks

Distributed object computing (DOC) middleware has been applied widely to domains ranging from telecommunications to aerospace, process automation, and e-commerce. DOC middleware shields developers from many distribution challenges and allows applications to invoke operations on target objects efficiently without concern for their location, programming language, OS platform, communication protocols and interconnects, and hardware [29]. Historically, however, many DOC middleware solutions have tightly coupled interfaces and implementations, which makes it hard to adapt to requirement

or environment changes that occur late in an application's life-cycle, *i.e.*, during deployment and/or at run-time.

Meta-programming mechanisms are techniques that help increase the flexibility and adaptability of applications, without degrading performance significantly. This paper describes two meta-programming mechanisms–*smart proxies* and *interceptors*– that we added recently to TAO, is an implementation of CORBA that is targeted for applications with high-performance and real-time QoS requirements. These two mechanisms allow CORBA applications to adapt to changing requirements or environmental conditions that occur late in an application's life-cycle without requiring obtrusive changes in existing software.

Based on our experience using smart proxies and interceptors to develop TAO applications, we have observed the following tradeoffs and limitations with smart proxies and interceptors:

Performance: Interceptors incur more overhead than smart proxies because they influence the processing of operations at multiple points along the invocation path. The portable interceptor results in Section 4.1.2 illustrate the overhead of supporting interceptors and the run-time costs of specific interceptor features.

In general, smart proxies perform better and consume less memory than interceptors. The smart proxy results in Section 4.1.1 show the circumstances where using smart proxies can improve performance. Even thought there is an extra layer of indirection, the overall performance can be improved by removing the gratuitous overhead of unnecessary remote invocations.

Generality: Interceptors can be applied to either servers or clients and can access operation-specific information. Therefore, they provide an effective meta-programming mechanism to handle advanced features, such as authentication and authorization, transparently end-to-end. In contrast, smart proxies only apply to specific interfaces accessed by clients. In particular, smart proxies can only influence the behavior at the beginning of an invocation.

Portability: Smart proxies are not currently part of the CORBA standard. Although many ORBs provide smart proxies as extensions, this feature is not portable. There is, however, a Portable Interceptors specification [7] that is being ratified by the OMG.

All the source code, documentation, and tests for TAO are open-source and can be downloaded from `www.cs.wustl.edu/~schmidt/TAO.html`.

Acknowledgements

Thanks to Brian Wallis <brian.wallis@ot.com.au> for helping with the design of TAO's smart proxy interface.

References

[1] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.3 ed., June 1999.

[2] M. Henning and S. Vinoski, *Advanced CORBA Programming With C++*. Addison-Wesley Longman, 1999.

[3] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[5] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom, "Flick: A Flexible, Optimizing IDL Compiler," in *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, (Las Vegas, NV), ACM, June 1997.

[6] C. Zimmermann, "Metalevels, MOPs and What the Fuzz is All About," in *Advances in Object-Oriented Metalevel Architectures and Reflection* (C. Zimmermann, ed.), Boca Raton, FL: CRC Press, 1996.

[7] Adiron, LLC, *et al.*, *Portable Interceptor Working Draft – Joint Revised Submission*. Object Management Group, OMG Document orbos/99-10-01 ed., October 1999.

[8] Object Management Group, *Security Service 1.8 Specification*, OMG Document security/00-11-03 ed., November 2000.

[9] O. Othman, C. O'Ryan, and D. C. Schmidt, "The Design and Performance of an Adaptive CORBA Load Balancing Service," *IEEE Distributed Systems Online*, vol. 1, December 2000.

[10] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrency and Distributed Objects, Volume 2*. New York, NY: Wiley & Sons, 2000.

[11] C. O'Ryan, F. Kuhns, D. C. Schmidt, O. Othman, and J. Parsons, "The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[12] T. Nakajima, "Dynamic Transport Protocol Selection in a CORBA System," in *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, (Newport Beach, CA), IEEE/IFIP, Mar. 2000.

[13] Object Management Group, *Extensible Transport Framework for Real-Time CORBA, Request for Proposal*. Object Management Group, OMG Document orbos/2000-09-12 ed., Feb. 2000.

[14] W. W. Ho and R. Olsson, "An Approach to Genuine Dynamic Linking," *Software: Practice and Experience*, vol. 21, pp. 375–390, Apr. 1991.

[15] N. Wang, D. C. Schmidt, and S. Vinoski, "Collocation Optimizations for CORBA," *C++ Report*, vol. 11, November/December 1999.

[16] K. Beck, *Smalltalk Best Practice Patterns*. Englewood Cliffs, NJ: Prentice-Hall, 1997.

[17] M. H. Austern, *Generic Programming and the STL*. Reading, MA: Addison-Wesley, 1999.

[18] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Using Principle Patterns to Optimize Real-time ORBs," *Concurrency Magazine*, vol. 8, no. 1, 2000.

[19] Object Management Group, *Minimum CORBA - Joint Revised Submission*, OMG Document orbos/98-08-04 ed., August 1998.

[20] J. A. Zinky, D. E. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, 1997.

[21] G. Kiczales, "Aspect-Oriented Programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[22] M. Shapiro, "Flexible Bindings for Fine-Grain, Distributed Objects," Tech. Rep. Rapport de recherche INRIA 2007, INRIA, Aug. 1993.

[23] D. Schmidt and S. Vinoski, "Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread-per-Object," *C++ Report*, vol. 8, July 1996.

[24] D. Schmidt and S. Vinoski, "Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread Pool," *C++ Report*, vol. 8, April 1996.

[25] D. Schmidt and S. Vinoski, "Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread-per-Request," *C++ Report*, vol. 8, February 1996.

[26] F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. Magalhaes, and R. Campbell, "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[27] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, "Using Interceptors to Enhance CORBA," *IEEE Computer*, vol. 32, pp. 64–68, July 1999.

[28] G. C. Hunt and M. L. Scott, "Intercepting and Instrumenting COM Application," in *Proceedings of the 5th Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.

[29] S. Vinoski, "New Features for CORBA 3.0," *Communications of the ACM*, vol. 41, pp. 44–52, October 1998.

Kava - Using Byte code Rewriting to add Behavioural Reflection to Java

Ian Welch and Robert J. Stroud
Department of Computing
University of Newcastle upon Tyne

Abstract

Many authors have proposed using byte code rewriting as a way of adapting or extending the behaviour of Java classes. There are toolkits available that simplify this process and raise the level of abstraction above byte code. However, to the best of our knowledge, none of these toolkits provide a complete model of behavioural reflection for Java. In this paper, we describe how we have used load-time byte code rewriting techniques to construct a runtime metaobject protocol for Java that can be used to adapt and customise the behaviour of Java classes in a more flexible and abstract way. Apart from providing a better semantic basis for byte code rewriting techniques, our approach also has the advantage over other reflective Java implementations that it doesn't require a modified compiler or JVM, can operate on byte code rather than source code and cannot be bypassed. In this paper we describe the implementation of Kava, our reflective implementation of Java, and discuss some of the linguistic issues and technical challenges involved in implementing such a tool on top of a standard JVM. Kava is available from http://www.cs.ncl.ac.uk/research/dependability/reflection.

1. Introduction

Many authors have considered the problem of reusing third party code in environments the developers did not originally consider [1, 2, 3]. For example, some proposals suggest ways to apply access control policies to code that has been developed without any thought for security [3]. Wrapping was originally proposed as a technique to enable adaptation of the code but it suffers from a number of problems such as identity confusion, or the self problem [1] etc. A solution to the problem is transform the code at the binary level [4]. This has proved to be a practical technique in the context of Java because the Java byte code retains a large amount of semantic information.

A number of byte code rewriting tools have been developed to ease the process of code rewriting. These include JOIE [5], Byte Code Engineering Library [6] and more recently Javassist [7]. Each toolkit provides object oriented representations of the structure of classes that can be used to rewrite classes on-the-fly.

The focus of these toolkits is on implementing changes to the behaviour of classes through programs that rewrite the class implementations. Users typically have to write programs that walk class structures and locate the appropriate places to make changes to the structure in order to implement some change to runtime behaviour. The actual implementation of changes this way is difficult for most programmers and highly error prone.

We argue that for applications where non-functional concerns are being implemented (security, transactions, debugging etc.) it would be more natural to specify changes to the behaviour of classes in terms of runtime abstractions.

For example, in order to trace state changes it would be more natural to redefine the runtime state access operation rather than manually write a program that walks all the methods of a class file and instruments pertinent field access operations.

Metaobject protocols and reflection are a good model for expressing such changes. Metaobject protocols provide abstractions of the runtime environment, and expose the protocols governing the execution in the runtime environment. Reflection means that changes to the implementation of these metaobject protocols will change the way in which code is executed at runtime.

We have implemented a highly portable implementation of a behavioural reflection for Java called Kava [8]. It provides a metaobject protocol for specifying changes to runtime behaviour and implements these changes through the use of structural rewriting toolkits such as JOIE, Byte Code Engineering Library, or Javassist. It is portable, is written entirely in Java, and unlike a number of other reflective Java implementations doesn't require a specialised Java Virtual Machine. Kava also provides support for properties such as strong non-bypassability and

```
public class TraceMethod implements Constants {
  private static String          class_name;
  private static ConstantPoolGen cp;
  private static int             out;     // reference to System.out
  private static int             println; // reference to PrintStream.println

  private static Method traceMethod(Method m) {
    Code   code  = m.getCode();
    int    flags = m.getAccessFlags();
    String name  = m.getName();

    // Create instruction list to be inserted at method start.
    String mesg = "tracing " + m.getMethodName();
    InstructionList patch  = new InstructionList();
    patch.append(new GETSTATIC(out));
    patch.append(new PUSH(cp, mesg));
    patch.append(new INVOKEVIRTUAL(println));

    MethodGen           mg  = new MethodGen(m, class_name, cp);
    InstructionList     il  = mg.getInstructionList();
    InstructionHandle[] ihs = il.getInstructionHandles();

    // First let the super or other constructor be called
    if(name.equals("<init>")) {
      for(int j=1; j < ihs.length; j++) {
        if(ihs[j].getInstruction() instanceof INVOKESPECIAL) {
          il.append(ihs[j], patch);  // Should check: method name == "<init>"
          break;
        }
      }
    }
    else
      il.insert(ihs[0], patch);

    // update stack size
    if(code.getMaxStack() < 2)
        mg.setMaxStack(2);

    return mg.getMethod();
  }

  public static void main(String[] argv) {
    JavaClass       java_class = new ClassParser(argv[i]).parse();
    ConstantPool    constants  = java_class.getConstantPool();
    cp = new ConstantPoolGen(constants);
    out     = cp.addFieldref("java.lang.System", "out",
                             "Ljava/io/PrintStream;");
    println = cp.addMethodref("java.io.PrintStream",
                              "println",
                              "(Ljava/lang/String;)V");

    Method[] methods = java_class.getMethods();
    for(int j=0; j < methods.length; j++)
      methods[j] = traceMethod(methods[j]);

    java_class.setConstantPool(cp.getFinalConstantPool());
    java_class.dump(class.getClassName()+".class");
  }
}
```

Figure 1 – Tracing Method Execution

reflection on inherited methods that other reflective Java implementations do not address.

In section 2 we discuss byte code rewriting and its shortcomings, in section 3 we introduce the Kava system, in section 4 we provide some examples of its application, in section 5 we discuss the implementation of Kava, in section 6 we provide an overview of related work and finally in section 7 we give our conclusions and outline future work.

2. Bytecode Rewriting

There are three main toolkits for rewriting bytecodes: Joie, Byte Code Engineering Library and Javassist. The first two toolkits provide object oriented frameworks for writing programs that manipulate the structure of class files. They provide loadtime representations of elements of class files such as methods, types, instructions etc. Java programs can then be written that describe how class files can be rewritten as late as load time. The main drawback with this approach is that the programmer has to have a detailed understanding of both the structure of class files and Java virtual machine programming. As the authors of Joie have observed, this makes it difficult for programmers to write reliable and easily understandable transformer programs. Javassist attempts to address this problem by providing a metaobject protocol for the rewriting of byte codes. It allows a programmer to work at a more abstract level. However, it sacrifices some of the power of the other toolkits without gaining a high enough level of abstraction. Also, it still requires the programmer to think in terms of reprogramming an existing implementation.

Figure 1 shows how the Byte Code Engineering Library can be used to trace method execution of a class.

This code adds a print statement at the start of each method. The `traceMethod` method generates the appropriate byte code for a print statement. While the main method traverses the structure of the class to locate the appropriate place to insert the instructions and finally ensure that the stack size after insertion is correct.

This process is obviously difficult for novice programmers to learn and is error prone. It is difficult as the code to be inserted is developed by hand and the programmer must manually add the appropriate entries to the constant pool. It is error prone because there is no separate type checking available for the code to be inserted. In addition to writing the code to be inserted the code for performing the insertion also has to be written from scratch every time and issues such as ensuring that the stack size is maintained correctly have to be addressed by the programmer.

To address these concerns, two improvements are needed:

- The ability to write the behavioural modifications in Java, and to be able to compile and verify these modifications as you would a normal class.
- The ability to declaratively specify where the behavioural modifications should be applied.

Kava provides these improvements. Behavioural adaptations are implemented using metaobject classes that can be compiled and verified, and the application of the metaobjects is driven by a binding specification that uses a declarative binding language.

3. Using Kava

In this section we introduce the basic concepts of behavioural reflection, and describe how Kava is actually used.

3.1. Behavioural Reflection

Reflection [9] is the process by which a system can reason about and act upon itself. A reflective system is composed of a base level and a meta level. The base level is the system being reasoned about, and the meta level has access to representations of the base level. *Reification* is the process by which the abstract representations of the base level are generated. A reflective system has the property that the meta level is *causally connected* to the base level. This means that changes at the meta level cause changes to the behaviour of the base level.

These notions of reflection have been extended to include the concept of the *metaobject protocol* [10] where an abstraction of the computation process and the protocols governing the execution of the program are exposed. A *metaobject* is bound to an object and controls the execution of the object. By changing the implementation of the metaobject the object's execution can be adjusted in a principled way. The protocols are implemented as methods of the metaobject.

Reflection and metaobject protocols have been successfully used to implement non-functional properties such

```
public interface IMetaObject {

  public void beforeExecuteMethod(IExecutionContext context)
  public void afterExecuteMethod(IExecutionContext context);
   /* called when a method is executed (including constructor/finalizer) */

  public void beforePutField(IFieldContext context);
  public void afterPutField(IFieldContext context);
   /* called when a field is accessed */

  public void beforeGetField(IFieldContext context);
  public void afterGetField(IFieldContext context);
   /* called when a field is read */

  public void beforeInvoke(IInvocationContext context);
  public void afterInvoke(IInvocationContext context);
   /* called when a method is invoked (including initialser */

  public void beforeException(IExceptionContext context);
  public void afterException(IExceptionContext context);
  /* called when an exception is thrown and caught */

}
```

Figure 2 – Interface for Kava MetaObject

as concurrent programming [11], atomic data types [12], fault tolerance [13], and security [14].

The Java programming language [15] includes a reflection package. This provides the ability to reify some aspects of the Java runtime environment such as methods, classes, fields, etc. and allows dynamic construction of proxies and dynamic method invocation. However, it does not provide the ability to modify the behaviour of an application through changes at a meta level. Kava provides powerful behavioural reflection without requiring changes to the Java Virtual Machine or requiring the use of source code preprocessing. It implements behavioural reflection through the principled rewriting of Java class files.

The Kava system allows each object or class to be bound to a metaobject. At the meta level runtime behaviours such as method invocation, method execution, field access, etc. can be redefined by the metaobject implementation. The metaobject implementation is constructed using reified aspects of the runtime object model. For example, a method is reified as an instance of a `Method` class.

The binding itself is described by a binding specification. This is written using a declarative binding language. Separating the binding information from the metaobjects increases the reusability of metaobjects as the bindings effectively parameterise the metaobjects. For example, a binding specification may bind a metaobject to different fields on different classes.

3.2 Using Kava

Each metaobject is an implementation of the interface `IMetaObject`. This defines a series of methods for intercepting and customising various aspects of the runtime behaviour of an object. See Figure 2 for the interface.

Each method has a *before* and *after* variant. The before methods are invoked before the behaviour, and the after methods are invoked after the behaviour. Each time a metaobject's method is invoked the behaviour's context is reified as an instance of a context object and passed as an argument. This makes the context accessible to the metaobject implementation. Some aspects of the context can be changed at the metalevel, such as the actual arguments passed to a method. On return to the base level the context object is converted back to the actual context of the behaviour.

Each before method can set the context such that the base level behaviour is overriden. This means that the base level behaviour will be suppressed. For example setting an override in a `beforeExecuteMethod` will result in the body of the method not being executed.

An example of a metaobject that implements the tracing of method executions similar to the example given in section 2 is:

```
public class MetaTrace
             implements IMetaObject {
  public void
     beforeExecuteMethod(
       IExecutionContext context) {
         System.out.println(
            "tracing " +
             context.getMethodName());
  }
}
```

In order to trace the methods of a particular class, it is necessary to establish a binding between instances of the class and instances of the `MetaTrace` class. These bindings are described using the Kava binding language in a special metaconfiguration file that drives the processing of a class by Kava. The binding specification shown below means that `MetaTrace` intercepts the execution of any method of the class `Test`.

```
<binding>
  <class>
    <classname>Test</classname>
    <metaclass>MetaTrace</metaclass>
    <intercept>
      <execute>
        <method>*</method>
        <parameters>*</parameters>
      </execute>
    </intercept>
   </class>
</binding>
```

If the implementation of `Test` is:

```
public class Test {
  public static
    void main(String[] args){
      (new Test).run(args[0]);
    }
  public void run(String s) {
      System.out.println("hello " + s);
  }
}
```

Then output of invoking the `run` method of `Test` with the actual parameter `World` is:

```
tracing run
hello World
```

Note that the code necessary to implement tracing behaviour is significantly more concise than the equivalent byte code transformation code. The metaobject that specifies the code to be invoked when a method is executed can also be compiled and verified therefore reducing the possibility of coding errors. The binding specification is significantly shorter than the code that traverses the class and inserts instructions at the appropriate place. Also, since it is a declarative specification it is easier to code and less likely to contain errors.

Kava is well suited to modifying the behaviour of classes where the interface of the class is not to be changed, or new keywords to be added to the language. As this example shows it is far more concise than an equivalent byte code transformation program, and it separates out the adaptation code (the metaobject) and the specification of where to apply the adaptation (the binding).

4. Examples

This section shows applications of Kava that highlight some of the more unusual features of the Kava metaobject protocol. Many implementations of reflective Java concentrate on intercepting method calls and tracing method calls is the standard example used to demonstrate a reflective system. Kava provides the ability to intercept the sending of method calls (invocation), field access, and exception handling in addition to the interception of method calls. The first example given here is of fine grained access control, this illustrates Kava's ability to control field access. The second example given here is how to prevent a particular type of denial of service attack, this illustrates Kava's ability to intercept the sending of method calls.

4.1 Fine grained access control

The Java programming language provides the following language level mechanisms for controlling access to class members such as methods or fields:

- Public access where code belonging to any class is allowed to access the member.
- Package access where access to the member is permitted only to code belonging to classes in the same package.
- Protected access where access to the member is permitted only to code that inherits from the functionality of the class.
- Private access where access to the member is permitted only to code that occurs in the body of the top level class that encloses the declaration of the member.

While this is adequate for a number of situations there is still the possibility that a more fine grained access control may be required for security purposes. For example, we may only want a certain field to be accessed by a limited number of classes that are spread across multiple packages.

Using Kava it is relatively simple to implement such a fine-grained scheme. In this example we focus on preventing access to fields by any but a small number of classes.

We implement the following protection metaobject MetaChkAccess that restricts access to a field to instances of two known classes GoodGuyA and GoodGuyB:

```
public MetaChkAccess implements
   IMetaObject {

  public void
    beforePutField(IFieldContext c) {
      checkAccess(c.getBase());
    }
    /* check any writes to the field */

  public void
    beforeGetField(IFieldContext c) {
      checkAccess(c.getBase ());
    }
    /* check any reads from the field */

  public void
    checkAccess(Object who) {
      if (who instanceof GoodGuy1 ||
        who instanceof GoodGuy2) {
        return;
      }
      else {
        // wrong class
        throw new
          SecurityException(
            "illegal access by  " +
            who.getClass().getName());
      }
  }
  /* check whether access is allowed */
}
```

The checkAccess method checks any access to a field. If a class other than one of the allowed classes attempts to access a field then a SecurityException is thrown. As SecurityException is a subclass of RuntimeException it does not have to be included in the declaration of the method.

The MetaChkAccess metaobject is then bound to any class that reads from or writes to the field protectedField of the class ProtectedClass by including the following in the binding specification:

```
<binding>
  <class>
    <classname>*</classname>
    <metaclass>MetaChkAccess</metaclass>
    <intercept>
      <getfield>
        <class>ProtectedClass</class>
        <field>ProtectedField</field>
      </getfield>
      <putfield>
        <class>ProtectedClass</class>
        <field>ProtectedField</field>
      </putfield>
    </intercept>
  </class>
</binding>
```

4.2 Denial of Service

Denial of service attacks are trivial to implement in Java. The simplest attacks consume resources by generating infinite numbers of objects such as windows that fill up the user's screen and occupy CPU time. Trivially this could be dealt with by defining a MetaRsrceLmt that watches how many instances of windows (all subclasses of java.awt.Frame) are created and limiting the number that can be created to a maximum:

```
public MetaRsrceLmt implements
  IMetaObject
{
  public void
    beforeInvoke(IInvocationContext c)
  {
    if (c.getTarget() instanceof
         java.awt.Window &&
        c.getMethod ().equals("<init>"))
    {
       maxCount++;
       if (maxCount > ARBITARY_MAX)
       {
          throw new
             RuntimeException(
     "exceeded max number of frames");
       }
    }
  }
}
```

This metaobject then would be bound to all method invocations by any method of any class:

```
<binding>
  <class>
    <classname>*</classname>
    <metaclass>MetaRsrceLmt</metaclass>
    <intercept>
      <invoke>
        <method>*</method>
        <parameters>*</parameters>
        <class>*</class>
        <targetmethod>
          <init/>
        </targetmethod>
      </invoke>
    </intercept>
  </class>
</binding>
```

A more sophisticated metaobject will allow the blocking of windows until one was destroyed, and would maintain a global count of windows and detect when windows were destroyed as well as created. However,

this example shows that resource creation can be easily controlled using a reflective approach.

5. Kava Implementation

5.1 Architecture

Kava is written purely in Java, it does not require any special Java Virtual Machine to work. The link between metaobjects and objects is realised by the rewriting of classes and addition of hooks into the class code. Figure 3 shows the Kava architecture. A classloader reads the class file as a stream of bytes. These can be retrieved from any source, normally from a file or from across the network. The classloader parses the byte stream and creates a JVM specific representation of a class. Normally this is passed to the verifier before it is instantiated by the JVM. However, *Kava* is used to intercept the byte stream before the classloader constructs the JVM specific class and applies the standard code transformations that realise control by metaobjects. As stated earlier Kava uses a binding specification file to determine what behaviours of what classes are to be brought under the control of particular metaobjects. It then adds traps into the code of the class to switch control when from the base level to the meta level (the associated metaobject) when the byte code base level objects carry out certain behaviours. After rewriting the class to include these traps, the classloader passes an internal representation of the class to the byte code verifier as before. This means that properties such as type safety are still honoured as before.

Note that the metaobjects are loaded by the classloader in exactly the same way as any other class, which means that they must satisfy the same security properties as any ordinary Java class. Metaobjects are ordinary Java classes and can be compiled which means that errors can be caught at an early stage.

Kava can be invoked either after a class is compiled or at the time the class is loaded into the JVM. In order to invoke Kava at loadtime a user-defined classloader must be used. In either case the traps that are added to the class are non-bypassable.

5.2 Instrumentation

The Kava metaobject protocol is implemented using the technique of byte code rewriting. Kava makes use of the Byte Code Engineering Library [6] toolkit to implement the standard transformations that add the hooks necessary to switch control from the base level to the meta level at runtime. Using a standard byte code rewriting toolkit frees us from dealing with technical details such as maintaining relative addressing when new byte codes are inserted into a method, or determining the number of arguments a method supports before it has been instantiated as part of a class.

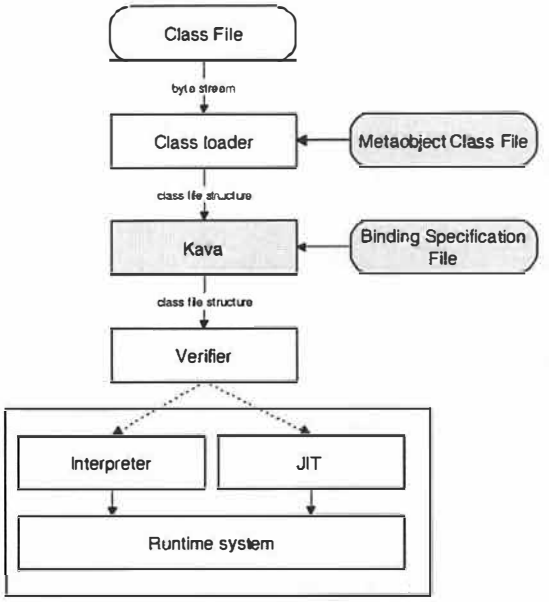


Figure 3 - Kava Architecture

Standard byte code rewritings are used to add hooks for individual methods and individual byte code instructions. These hooks reify the context of a behaviour that is being trapped, invoke the metaobject associated with an object and reflect any changes to the context back to the base level. The metaobjects that are invoked are completely separate from the byte code hooks and are developed entirely in Java. This separation means that the runtime meta level can be adjusted dynamically at runtime although which behaviours are trapped is determined at loadtime.

For example, returning to the example of section 3.2 the class `Test` included the following `run` method:

```
public void run(String s) {
  System.out.println("hello " + s); }
```

After the metaobject `MetaTrace` is bound to `Test` using the binding specification presented earlier, the `run` method is effectively rewritten by Kava as:

```
public void run(String s) {
  Context c = new Context
    (this, "run", "void",
     "java.lang.String", new Object[]
     {s});
  getMeta().beforeExecuteMethod(c);
  if (!c.override()) {
  System.out.println("hello " +
     (String)c.getArg(0));
```

```
        getMeta().afterExecuteMethod(c);
    }
}
```

The code in **bold** has been added by Kava.

First, at the beginning of the method block a context object that represents the invocation frame is created. It contains a pointer to the base level object itself, the name of the method being executed, the return type, the types of the parameters, and the actual parameters marshalled into an array of objects. Then the metaobject associated with the base level object is retrieved using a method added earlier by Kava and the `beforeExecuteMethod` method invoked. In this case `getMeta()` returns a pointer to an instance of a `MetaTrace` so the method name is printed out. Following the invocation of `beforeExecuteMethod` the arguments passed within the context object are unpacked, and the base level code is invoked. Finally, at the end of the method block the `afterExecuteMethod` method of the associated metaobject is invoked. In this case there was no implementation of the method so nothing occurs at the meta level.

Since a metaobject method may override the corresponding base level behaviour we add an `if ... then` clause. This ensures that when an override is indicated then the base level behaviour is suppressed.

This is an example of standard transformation for a block of code. The transformation for intercepting behaviour such as setting the value of a field is very similar but finer-grained with the hook code being around a single instruction.

5.3 Binding language

As explained in section 5.1 the binding specification file determines where Kava introduces the hooks into the base level code. The concept is that to make metaobjects more reusable the binding should be specified completely separately of both the base and meta level.

The binding specification contains multiple base object and metaobject class bindings. Each binding is between one class and a metaobject class. For that binding the particular behaviours to be brought under the control of the metaobject are specified, for example the execution of methods, or the setting of fields. These are parameterised by information such as the name of the field or method, the type of the target (in the case of setting a field, or invoking a method) etc.

5.4 Special Features

In this section we give an overview of some of the special features supported by Kava: strong encapsulation, reflection on inherited methods, exception handling and context objects.

5.4.1 Strong encapsulation

One of the benefits of the Kava implementation is its support for strong encapsulation. Strong encapsulation is the property that it is difficult to bypass the metaobject bound to the base level object. This has been achieved by avoiding the use of a separate wrapper class. Since hooks are added directly into method bodies we greatly reduce the possibility that the hooks could be bypassed. This is because there is no way to express in the Java language a branching to an arbitrary point in a method body.

It is true that if a malicious code transformer rewrote a class file that was pre-processed by Kava then our hooks could be removed. However, this can be easily guarded against through the use of a mixture of operating system protection and the use of code signing techniques.

5.4.2 Inherited Methods

When a Java class inherits a method from its superclass the bytecode implementing the method is not reproduced in the implementation of the class. For example, if class `C` inherits the `run` method from class `D` then the byte code for `run` is still to be found in `D`'s class file not `C`'s class file. If we bind `C` to a metaobject metaclass `MC`, and try and bring the execution of `run` under the control of `MC`, Kava will fail because it cannot find the byte code implementation of run. The obvious answer is to bind `MC` to `D` as well and add the hooks into `D`'s run method. However, we may not want `D` to be brought under the control of `MC`, indeed we may even want it to be bound to an entirely different metaclass.

The answer to this problem is to add a method `getMeta` to each base level class that returns a pointer to the metaobject bound to the base level object. We then ensure that superclass methods inherited by classes that are bound to a metaobject have hooks added to them that use the `getMeta` method to determine which metaobject to invoke.

When an instance of `C` has its inherited method `run` invoked the JVM's dynamic resolution of method calls will mean that the `getMeta` method appropriate to `C` will be invoked. This means that the metaobject bound to `C` will be returned.

When an instance of D had its method run invoked, the JVM's dynamic resolution of method calls will mean that the getMeta method appropriate to D will be invoked. This means that the metaobject bound to D will be returned.

This approach ensures that the correct metaobject is invoked in both cases. The approach taken here is similar to that found in [3].

5.4.3 Exception Handling

Kava allows the raising and throwing of exceptions to be intercepted and handled by the metaobject bound to an object. The beforeException method is invoked before an exception is thrown at the base level. It allows the exception throwing to be overridden, this might be necessary where the metaobject is implementing distribution at the meta level and the exception has to be propagated to a remote client. The afterException method is invoked after an exception has been thrown or raised at the base level. It doesn't allow overriding of this behaviour but does allow additional processing to take place such as the propagation of the exception to related objects if a number of objects are co-operating and need to be aware of each other's status.

5.5 Context Objects

Kava uses the concept of Context objects to simplify the metaobject protocol and also allow the possibility of lazy reification. The metaobject sometimes will need access to runtime instances of Method, Class or Field. However, generating these is a relatively expensive process so we defer their creation by passing the minimum information needed to derive them in a Context object. As part of the Context interface we provide methods for generating the reified instances. The standard Java reflective API is used to generate these instances. In the future we would like to apply the same technique to the actual parameters passed to the metaobject.

5.6 Performance

We have made some preliminary measurements of the performance of Kava. They indicate that the most expensive operation is the generation of the context. Presently, this expense is more than doubling the execution speed of a number of instructions. We are currently exploring two main approaches to improving performance. The first approach is to use caching of context information, and the second is to allow selective reification.

6. Related Work

In this section we briefly review a number of other reflective Java implementations and attempt to categorise them according to the point in the Java class lifecycle that reflection is implemented.

The Java class lifecycle is as follows. A Java class starts as source code that is compiled into byte code, it is then loaded by a class loader into the Java Virtual Machine (JVM) for execution, where the byte code is further compiled by a Just-In-Time compiler into platform specific machine code for efficient execution.

Different reflective Java implementations introduce reflection at different points in the lifecycle. The point at which they introduce reflection tends to characterise the scope of their capabilities. In order to bring the base level under the control of the meta level the base level system is modified through the addition of traps. These traps are known as meta level interceptions [16]. For example, in Reflective Java method calls sent to the base object are brought under control of an associated metaobject by trapping each method call to the baseobject. This is done by pre-processing the source code of the base level class. A contrasting example is MetaXa where the traps are in the implementation of the dispatch mechanism of the Virtual Machine. As the traps exist in the Virtual Machine itself, the source code of classes to be made reflective is not required. However, unlike Reflective Java, a specialised JVM must be used.

Table 1 summarises the features of various reflective Java implementations. All these implementations have drawbacks that make them unsuitable for use with compiled components or in a standard Java environment where the purpose is to add security. Some require access to source code, and others are non-standard because they make use of a modified Java platform.

Point in Lifecycle	Reflective Java	Description	Capabilities	Restrictions
Source Code	Reflective Java [17]	Preprocessor.	Dynamic switching of metaobjects. Intercept method invocations.	Can't make a compiled class reflective, requires access to source code.
Compile Time	OpenJava [18]	Compile-time metaobject protocol.	Can intercept wide range of operations, and extends language syntax.	Requires access to source code.
Byte Code	Bean Extender [19], Dalang [20], JavaAssist [7]	Byte code preprocessor (Bean Extender), byte code rewriting as late as load time (Dalang, JavaAssist).	No need to have access to source code.	Bean Extender – restricted to Java Beans. Dalang, and Javassist – limited capabilities. requires offline preprocessing.
Runtime	MetaXa [21], Rjava [22], Guarana [23]	Reflective JVMs.	Can intercept wide range of operations. Can be dynamically applied.	Custom JVM.
	java.lang. reflect [15]	Reflective capabilities part of the standard Java development kit.	Runtime introspection, dynamic dispatch and on-the-fly generation of proxies.	Overall introspection rather than behavioural or structural reflection.
Just-in-time Compilation	OpenJIT [24]	Compile-time metaobject protocol for compilation to machine language.	Can take advantage of facilities present in the native platform. No need for access to source code. Dynamic adaptation.	Custom Just-in-time compiler.

Table 1 - Reflective Java Implementations

In contrast, Kava does not require access to source code because it is based on byte code rewriting, doesn't require a non-standard Java environment and provides a rich set of capabilities. It also provides what we refer to as *strong encapsulation*. Most implementations add traps through renaming of classes, or renaming methods, which means that it may be possible to call the original methods and therefore bypass the meta layer. Kava actually adds the traps directly into the method bodies avoiding this problem. Dalang was an earlier implementation of a loadtime reflective Java we implemented that suffered from this problem. See [20] for an account of the evolution of Kava from Dalang.

The closest reflective Java to Kava is a behavioural reflection add-on implemented as a demonstration of the capabilities of JavaAssist, a byte code rewriting tool based on structural reflection. Like Kava, this add-on adds hooks to the classes using byte code rewriting and has a similar meta level architecture of a binding between an object and a metaobject. However, it does not provide reflection on static members, on method invocation, or exception raising. Also it doesn't support

reflection on methods that have been inherited from a superclass, nor does it support the concept of a binding specification.

7. Conclusions and Future Work

Kava focuses on the behavioural changes programmers want to impose on third-party code instead of the messy structural changes that byte code transformation tools deal with. Kava allows adaptations to be developed, compiled and tested independently of the target code, then declaratively combined with the target code. This reduces the chance of error and makes that task of adapting the behaviour of third-party code more tractable.

Kava implements behavioural reflection in Java using byte code transformation as the underlying technique. This approach has allowed the creation of a tool unlike other reflective Java implementations is portable and can bring reflect on a wide range of runtime behaviours.

Kava is available for download from http://www.cs.ncl.ac.uk/research/dependability/reflection. We are currently in the process of tuning the implementation to support lazy reification of context objects. We are also investigating the application of Kava to a case study based on flexible security for an enterprise modelling system.

Acknowledgements

This work has been supported by the UK Defence Evaluation Research Agency, grant number CSM/547/UA and also the ESPIRIT LTR project MAFTIA.

References

[1] U. Holzle, "Integrating Independently-Developed Components in Object-Oriented Languages," ECOOP'93, Kaiserslautern, Germany, 1993.

[2] G. Czaijkowski and T. v. Eicken, "JRes : A Resource Accounting Interface for Java", OOPSLA'98, 1998.

[3] R. Pandey and B. Hashii, "Providing Fine-Grained Access Control for Java Programs," ECOOP'99, Lisbon, Portugal, 1999.

[4] R. Keller and U. Holzle, "Binary Component Adaptation," ECOOP'98, 1998.

[5] G. A. Cohen and J. S. Chase, "Automatic Program Transformation with JOIE," USENIX Annual Technical Symposium, New Orleans, Louisiana, 1998.

[6] M. Dahm, "Byte Code Engineering with the JavaClass API," Friei Universitat, Berlin, Technical Report B-17-98, 1998.

[7] S. Chiba, "Load-time Structural Reflection in Java," European Conference on Object-Oriented Programming, 2000.

[8] I. Welch and R. J. Stroud, "Kava - A Reflective Java Based on Bytecode Rewriting," in *Reflection and Software Engineering*, vol. 1826, W. Cazzola, R. J. Stroud, and F. Tisato, Eds. Heidelberg, Germany: Springer-Verlag, 2000, pp. 157-169.

[9] P. Maes, "Concepts and Experiments in Computational Reflection," OOPSLA'87, Orlando, Florida, 1987.

[10] G. Kiczales, J. des Rivieres, and D. G. Bobrow, *The Art of the Metaobject Protocol*: Massachusetts Institute of Technology, 1991.

[11] S. Matsuoka, T. Watanabe, and A. Yonezawa, "Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming," ECOOP'91, 1991.

[12] R. J. Stroud and Z. Wu, "Using Metaobject Protocols to Implement Atomic Data Types," ECOOP'95, Aarhus, Denmark, 1995.

[13] J.-C. Fabre, V. Nicomette, T. Perennou, Z. Wu, and R. J. Stroud, "Implementing Fault-tolerant Applications using Reflective Object-Oriented Programming," FTCS-25, Pasadena, USA, 1996.

[14] M. Benantar, B. Blakley, and A. J. Nadain, "Approach to Object Security in Distributed SOM," *IBM Systems Journal*, vol. 35, 1996.

[15] Sun Microsystems Inc., "Java Development Kit 1.3.0 Documentation," , 2000.

[16] C. Zimmerman, "Metalevels, MOPs and What all the Fuzz is All About," in *Advances in Object-Oriented Metalevel Architectures and Reflection*, C. Zimmermann, Ed.: CRC Press, 1996.

[17] Z. Wu and S. Schwiderski, "Reflective Java : The Design, Implementation and Applications," , 1996.

[18] M. Tatsubori and S. Chiba, "Programming Support of Design Patterns with Compile-time Reflection," Workshop on Reflective Programming in C++ and Java, 1998.

[19] IBM, "Bean Extender Documentation, version 2.0", 1997.

[20] I. S. Welch and R. J. Stroud, "From Dalang to Kava - the Evolution of a Reflective Java Extension," Second International Conference on Meta-Level Architectures and Reflection, Saint-Malo, France, 1999.

[21] M. Golm, "Design and Implementation of a Meta Architecture for Java", M.Sc. Erlangen, 1997.

[22] J. de. O. Guimarães, "Reflection for Statically Typed Languages" ECOOP'98, 1998.

[23] A. Oliva, and L. E. Bizato, "The Design and Implementation of Guaraná,", COOTS'99, 1999

[24] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and Y. Kimura, "OpenJIT: An Open-Ended, Reflective JIT Compiler Framework for Java," ECOOP'2000, 2000.

Content-Based Publish/Subscribe with Structural Reflection*

Patrick Th. Eugster Rachid Guerraoui
Communication Systems Department
Swiss Federal Institute of Technology, Lausanne
{Patrick.Eugster, Rachid.Guerraoui}@epfl.ch, http://www.d-a-c-e.com

Abstract

This paper presents a pragmatic way of implementing content-based publish/subscribe in a strongly typed object-oriented language. In short, we use structural reflection to implement filter objects through which applications express their subscription patterns. Our approach is pragmatic in the sense that it alleviates the need for any specific subscription language. It preserves encapsulation of message objects and helps avoiding errors. We illustrate our approach in the context of Distributed Asynchronous Collections (DACs), programming abstractions for message-oriented interaction. DACs are implemented in Java, whose inherent reflective capabilities fully satisfy the requirements of our content-based subscription scheme. Our approach is however not limited to the context of DACs, but could be put to work easily in other existing event-based systems.

1 Introduction

Publish/subscribe in perspective. The importance of flexible, well-structured, but especially scalable communication mechanisms has been drastically increasing in the last decade. Applications tend to become very dynamic, i.e., components are not always up and are not locality-bound. These constraints visualize the demand for more flexible communication models, reflecting the nature of tomorrows applications. The *publish/subscribe* interaction style has proven its ability to fill this gap. Based on the concept of *information bus* [OPSS93], publish/subscribe promotes the *decoupling* of parties in *time* as well as *space*:[1] consumers *subscribe* to the information bus by specifiying the nature of the information they are interested in, and producers publish information on that bus.
The classical *topic-based* or *subject-based* publish/subscribe style involves a classification of the information by introducing group-like notions [Pow96], and is incorporated by most industrial strength solutions, e.g., [Cor99, TIB99, Ske98, AEM99]. Topics are however static and allow only a limited *expressiveness* [Car98]. More recently, research efforts have been targeted towards *content-based* (*property-based* [RW97]) publish/subscribe schemes [Car98, SA97, BCM+99]. This more flexible variant removes entirely the "arbitrary" division of the information space, and lets consumers delineate their individual interests by expressing *properties* of messages they wish to receive.

Current practice. Common event-based systems relying on the content-based publish/subscribe paradigm equate *properties* of messages to *attributes* of those messages. In most cases, a subscription language is used to express ranges of values for those attributes, which violates object encapsulation: a subscription *pattern* [2] expressed with such a subscription scheme exposes the message's *state*, and the resulting *filter* queries messages by accessing their *attributes*. Furthermore, subscription languages can not be extended or customized by the application developer, they are orthogonal and redundant with the programming language,[3] and they are very er-

*This work is partially supported by Agilent Laboratories and Lombard Odier & Co.

[1] Time decoupling: the interacting parties do not need to be up at the same time. Space decoupling: the interacting parties do not need to know each other.

[2] In [Car98], the notion of *pattern* is used in a different sense, namely to express *event-correlation*: a notification is triggered upon arising of a combination of several elementary events.

[3] A similar mismatch has been largely discussed in the domain of object-oriented databases, where two separate languages coexist; one for the *definition* of data and another one for the *querying* of data [BZ87].

ror prone: syntax errors violating the subscription grammar are only seized at runtime when a pattern is parsed, just like syntaxically correct constraints based on badly written attributes.

Filter library. Our approach which has been realized in the context of DISTRIBUTED ASYNCHRONOUS COLLECTIONS (DACs) [EGS00a] – simple JAVA programming abstractions which encompass different message-oriented interaction styles – avoids any subscription language, and respects encapsulation. It promotes the expression of subscription patterns by combining general-purpose *filter objects*. These filter objects preserve encapsulation by querying message objects through methods which are dynamically defined by the application, along with the semantics of the evaluation of the invocation results. The subscription grammar is inherently expressed through the resulting API, which strongly reduces the number of runtime errors. Filters are thus pictured as first class citizens, and their implementation relies on *structural reflection* [Coi87] of the message objects.

Structural reflection. As pointed out in [Fer89], there are mainly two kinds of reflection. The *computational* (*behavioral*) reflection is concerned with the *reification* of computations and their behaviour. In contrast, *structural* reflection reifies the structural aspects of a program, such as data types.
As we will show in this paper, structural reflection can be used to express subscription patterns in content-based publish/subscribe the same way it has already been used in object-oriented data management systems to express object queries (e.g., [SO95]). In our particular context, structural reflection can be reduced to a single aspect: the capability of *representing* structures of objects. This is sufficient to dynamically define the methods that our filters must use to query message objects. The *introspection* capabilities of JAVA [Sun99a] offer sufficient support for this, and the possibility of *modifying* data structures is not required.

Contributions. This paper presents how we have realized content-based publish/subscribe in our DAC framework for distributed computing, which is implemented in JAVA on UNIX. We illustrate how our approach (1) circumvents the need for any subscription language, (2) preserves object encapsulation, and (3) helps avoiding type errors. We discuss the flexibility/performance trade-off introduced by our use of reflection by outlining the optimizations we have applied, such as runtime generation of static code from dynamically defined filters.

Roadmap. This paper is structured as follows: Section 2 overviews the limitations of existing approaches to expressing content-based subscription patterns. Section 3 presents our approach to content-based publish/subscribe based on structural reflection. In Section 4 we illustrate the use of our subscription scheme through a small example. Section 5 discusses performance issues. Section 6 highlights alternative approaches. Section 7 concludes with final remarks.

2 Approaches to Content-Based Publish/Subscribe: Background

Content-based publish/subscribe removes limitations of the static topic-based flavor, but suffers from the dynamism it introduces. Besides making the reuse of existing multicast primitives problematic [OAA+00], content-based publish/subscribe is hard to express in an object-oriented setting. In this section, we illustrate the latter difficulty by outlining the limitations of existing content-based schemes.

2.1 Subscription Languages

In content-based publish/subscribe, subscription languages are the most commonly used means of describing subscription patterns. Such languages can be based directly on the *attributes* of the described objects or on additional *properties* attached to those objects. By viewing asynchronous invocations as events, the *arguments* of such invocations can be used as matching criteria.

Attributes. In systems like SIENA [Car98], ELVIN [SA97] or GRYPHON [BCM+99, SBS98],[4] event notifications are viewed as flat structures,

[4] In GRYPHON, reflection is also used ([SBS98]), but not for the expression of subscription patterns: the GRYPHON system uses the same information dissemination mechanisms

i.e., *records* with several *fields*. A subscription language is used to impose ranges of values for those fields. Figure 1 outlines this concept schematically. Relying on *attribute-value* pairs enables very efficient realizations, since computational overhead is reduced by directly accessing attributes. This approach however bears several dangers:

Violation of object encapsulation: In the example outlined in Figure 1, the `from` attribute is used as subscription criterion, and is consequently directly accessed when the object is queried.

Errors: Syntax errors violating the subscription grammar are only seized once a pattern is parsed, i.e., at runtime. Another more malign type of errors result from badly typed attribute names. Subscription patterns containing such errors do not violate the syntax grammar, and might remain undetected without type checks.

Learning phase: Subscription syntaxes are often very complex and used with a single publish/subscribe middleware. This reduces portability of applications.

To increase portability of applications some engines implement standardized API's like the OMG's CORBA NOTIFICATION SERVICE [OMG00], which repairs certain lacks [SV97] of the CORBA EVENT SERVICE [OMG98]. Among the new features in [OMG00] are a content-based subscription scheme based on a simplified kind of typed events, replacing the typed events of the ancestor. These *structured events* are roughly composed of two types of fields, namely (1) fixed fields and (2) variable fields consisting of *name-value* pairs, to which applications map their specific needs. The fields of messages are seen as their attributes and are directly accessed through *filter objects* for content-based filtering – violating encapsulation. Patterns are expressed by strings following the DEFAULT FILTER CONSTRAINT LANGUAGE, a complex subscription language which extends the TRADER CONSTRAINT LANGUAGE.

Properties. The SUN counterpart to the CORBA NOTIFICATION SERVICE is the JAVA MESSAGE SERVICE (JMS) API [HBS98]. The JMS covers topic-based publish/subscribe

it offers to applications (which is its primary concern) for internal protocol communication.

Message m	`public class ChatMsg {`
	`public String from;`
	`...`
	`}`

Criteria	"message sent by Tom"
Argument	`String criteria = "from is Tom";`
Evaluation	`m.from.equals("Tom")`

Figure 1: Subscription Language

(*all-of-n*) as well as *message queuing* (*one-of-n*) [BHL95, DEC94, Sys00, Mic97]. Content-based filters can be applied with both interaction schemes. The filtering is based on attributes of the message headers, and on *properties* (name-value pairs), which are explicitly attached to message objects. Subscription patterns are expressed as JAVA strings. The specification includes a subscription grammar that these strings must respect.
Properties explicitly attached to message objects are artificial and in practice strongly redundant with the information carried by those objects. In many cases, the properties are faithful duplicates of the attributes of the message objects, which leads to violating encapsulation.

Arguments. MICROSOFT's COM+ [Obe00] promotes a model similar to the abandoned typed model of the CORBA EVENT SERVICE. Asynchronous invocations are viewed as events, but latter ones are not reified. The primary filtering is thus made on the types of the subscribers, as illustrated by Figure 2. By viewing an invocation as an event, the invocation arguments can be viewed as the attributes of the resulting notification. Filters in COM+ are expressed on invocation arguments through a limited subscription grammar. Encapsulation seems to be preserved by avoiding the reification of events.

Subscriber s	`public class Chatter {`
	`public void in(String from, ...);`
	`...`
	`}`

Criteria	"message sent by Tom"
Argument	`String criteria = "from is Tom";`
Evaluation	`from.equals("Tom")`

Figure 2: Events vs. Invocations

2.2 Template Objects

The JAVASPACE specification [Sun99b] (inspired by LINDA's TUPLE SPACE [Gel85]) adopts an approach based on *template objects*.
When subscribing to a JAVASPACE, a subscriber provides a template object `t`. A message object `m` is only delivered to that subscriber if `m` conforms to the type of `t`, and if every attribute of `t` which is not `null` references an object equal to the corresponding attribute of `m` (cf. Figure 3). Equality is tested by comparing byte-wise the two objects in marshalled form. As shown by [FHA99], this approach represents a very convenient subscription scheme which can be put to work easily. However, encapsulation is violated, and there are certain limitations in expressiveness:

Limited comparisons: Attributes are compared for strict equality, and it is not straightforward to express a range (discrete or not) of possible values for an attribute.

Limited granularity: In JAVA, an attribute can reference an object, which itself has attributes, etc. Attributes of JAVASPACE entries are however matched as a whole. This limitation is also found with most of the previous approaches based on subscription languages.

Limited combinations: By providing a template object `t`, a subscriber will receive every object `m` whose attributes *all* match the attributes of `t`. It is thus difficult to express alternatives (*or*) on different attributes.

Limited values: Since `null` is chosen to play the role of wildcard, attributes can not be of native types, and `null` can not be easily used as a concrete value for an attribute. For each such attribute [Sun99c] proposes to add an additional boolean attribute to indicate a `null` value.

3 Reflection-Based Publish/Subscribe

In this section we present our approach to specifying content-based subscription patterns. It is based on structural reflection of message objects and avoids limitations stated in the previous section.

Message m	`public class ChatMsg {`
	`  public String from;`
	`  ...`
	`}`

Criteria	"message sent by Tom"
Argument	`ChatMsg mt = new ChatMsg();` `t.from = "Tom";`
Evaluation	`m.from.equals(t.from)`

Figure 3: Template Object

3.1 Overview

Roughly spoken, the application programmer *defines* conditions on message objects, by specifying *methods* through which these objects should be queried, along with expected *values* that are *compared* to the values returned by invoking these methods.
Subscription patterns are expressed through an API, which inherently expresses a subscription grammar: by instantiating and combining filter objects, syntax errors violating the grammar are detected by the JAVA compiler. Thanks to the structural reflection of message objects, type errors are avoided by verifying the methods specified by the application.
In the following, we present our customizable filter objects, called *conditions*. These enable the dynamic definition of conditions on message objects, and are realized in a general manner through *accessors*, which we introduce first.

3.2 Accessors

Accessors are specific objects used to access partial information on the runtime message objects.

Querying objects. Informally, an accessor A is characterized by a set of tuples: $A = ((M_1, P_1), ..., (M_k, P_k))$, where every M_i is a method and $P_i = (P_{i,1}, ..., P_{i,i_l})$ its corresponding argument list. Whenever a method M_i is applied to an object, this subsumes that it is invoked with its arguments P_i.
An accessor can be seen as a function, which applied to a message object returns another object: $A(o : obj) \rightarrow obj$. When such an accessor A is evaluated for a message object m, M_1 is invoked on m

and every method M_{i+1} $(0 < i < k)$ is recursively invoked on the result of M_i. Finally, the result of M_k is returned.[5]

In JAVA, an accessor object implements the interface `Accessor` given in Figure 4, and is evaluated by calling the `get()` method with the message object as argument. This method can throw exceptions raised when evaluating the method chain, which enables the reaction to exceptions. Returning `null` in case of exceptions would contradict the use of `null` as matching criterion.

```
public interface Accessor {

  public Object get(Object m) throws Exception;
}
```

Figure 4: `Accessor` Interface

Using Java reflection. To implement our accessors, we rely on structural reflection. The inherent JAVA language reflection capabilities [Sun99a] consist in a type-safe API that supports *introspection* about classes and objects in the current JAVA VM at runtime. We view introspection as one aspect of structural reflection, limited to the reification (in the sense of *representation*) of *structures* of types and classes at runtime. A second aspect, the *modification* of those structures is, like computational reflection, not addressed by the JAVA core reflection API.[6]

In short, JAVA provides *meta-objects* which reify classes, methods, fields, constructors, etc. We make extensive use of meta-objects for methods (`java.lang.reflect.Method`) to reify the M_i's of accessors. This defers to runtime the choice of *which* method is to be invoked, and enables also to effectively perform such a *dynamic invocation*.[7] We avoid using objects reifying attributes (`java.lang.reflect.Field`), since dealing with them means abandoning encapsulation.

[5] With $k = 0$, the object m itself is accessed as a whole. $i_l = 0$ means that M_i is an argument-less method. We do not consider side-effects of the access methods M_i.

[6] JAVA 1.3 integrates a limited mechanism for computational reflection with the `java.lang.reflect.Proxy` class.

[7] Note that with JAVA method objects, a native value is wrapped by an instance of its corresponding object type, which makes the nesting of invocations even simpler.

```
public final class Invoke
  implements Accessor, java.io.Serializable
{
  /* only one method, can be null */
  public Invoke(Method M, Object[] args) {...}
  /* with nested accessor */
  public Invoke(Accessor nested, Method M,
                Object[] args) {...}
  /* structurally conformant objects, nesting */
  public Invoke(String methodNames,
                Object[][] args) {...}
  ...
  public Object get(Object m)
                  throws Exception {...}
}
```

Figure 5: Invoke Class (Excerpt)

Specifying methods. We have used JAVA reflection for the implementation of the `Invoke` class shown in Figure 5, a general-purpose accessor. The first constructor enables the expression of a single method invocation. The other constructors shown in the figure enable the creation of an accessor reflecting nested method calls; by specifying an explicitly created nested accessor, or by specifying the names of the methods to be invoked. This adduces the two ways for an application to specify a method:

By method object: The application explicitly deals with reflection, and provides a `Method` object. As explained in [BW98], JAVA enforces *name equivalence* of types, and a method object `M` is therefore bound to a single type `T`: if a method `M` for type `T` is applied to an object `m` which does not conform to `T`, `null` is returned – even if `m` implements a method of the same name and signature than `M`. By specifying methods as objects, the application implicitly defines the type of message objects it is interested in.

By method name (and signature): Specifying the name of a method and its arguments (and implicitly the method's signature) can be interesting to enforce *structure equivalence* of types, i.e., subscribing to *all* objects which implement a given method, independently of their type. This implies, for *each* evaluated message object, a *dynamic lookup* of the corresponding method object (through `java.lang.Class`) by the accessor.

In Section 5 we evaluate the two possibilities in terms of efficiency, and show that the knowledge of the type of message objects is important for performance optimizations.

Avoiding type errors. Knowing the type of the fitting message objects is also useful for type checking. If all methods of an accessor are reified, the return type of each such M_i can be checked for its conformance to the type bound to M_{i+1}. Similarly, the type of each provided argument $P_{i,j}$ can be checked for its conformance to the type of the j-th formal argument of M_i. By enforcing these checks, the `Invoke` class rules out type errors.[8] To enforce such checks without explicit use of reflection, message object types can also be specified by their name. This is illustrated in Section 4 through a small programming example.

3.3 Conditions

While a message object is queried through an accessor, a condition object evaluates the obtained information, i.e., decides whether it represents a desirable value.

Model. A condition $C = (A, R, B)$ represents a single condition that a message object m must fulfill in order to be delivered. B is a comparison function which can be viewed as a binary predicate: $B(o_1 : obj, o_2 : obj) \dashrightarrow bool$. The two arguments are (1) a predefined result R and (2) the result of the invocation chain represented by the accessor A. A condition is thus evaluated against a message object m, and evaluates positively iff m satisfies that condition: $C(o : obj) \dashrightarrow bool$, and $C(m) = B(R, A(m))$. Figure 6 outlines the different evaluation stages of a condition. A similar scheme can be found for object queries in object-oriented data management systems, e.g., TIGUKAT [SO95].[9]

[8] To verify whether a given reified type conforms to another one, we mainly rely on the `isAssignableFrom()` method in class `java.lang.Class`.

[9] The major difference between queries in an object database and the filtering of messages by a middleware is the *duration* of a query. With a middleware system based on content-based publish/subscribe, the query is expressed for *future* objects. In object databases, queries are performed on a snapshot of the system, but the expression of the query can be made similarly. [PO93] also describes the use of reflection for a closer integration of the language with TIGUKAT.

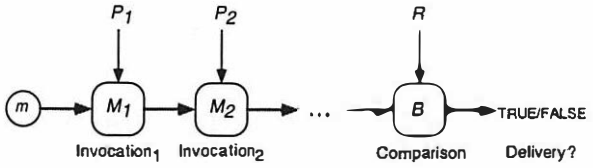


Figure 6: Applying a Condition to a Message Object

Basic conditions. In JAVA, a condition object implements the `Condition` interface given in Figure 7. It is evaluated for a given message object `m` by invoking `conforms()` with `m` as argument. The condition classes we propose are conceptually similar to the *predicates* found in JGL [Obj99] that are used in conjunction with centralized collections. The main difference is that our condition objects are specific to publish/subscribe, by representing queries on *future* objects.
What differentiates our condition classes are the comparison functions they encapsulate. The other attributes, namely accessor and result, are initialization arguments and can thus be factored out.

```
public interface Condition {

  public boolean conforms(Object m);
}
```

Figure 7: `Condition` Interface

Comparisons. JAVA inherently defines three basic comparison mechanisms, which can be considered as candidates for B:

I. **Is object `o1` identical to object `o2`?**
The comparison of two objects with the `==` operator yields `true` iff the two arguments are references to the same object. This comparison is less useful in our context, since two compared objects usually originate from different VMs. By default two such objects are never identical.

II. **Is object `o1` equal to object `o2`?**
Every object can also be compared to any other object by means of the `equals()` method, which is inherent to all JAVA objects and can be overwritten by application-defined classes.

III. **How does object o1 compare to object o2?**
This is for objects implementing the `java.lang.Comparable` interface,[10] providing a method `compareTo()`. The return value is an integer, indicating the order of the object `o1` with respect to `o2`. Such objects manifest a natural ordering, e.g., class `java.lang.Integer`, and can thus be matched against a range of values. Comparisons can be moved out of the compared objects by using `java.util.Comparator` objects, which are binary predicates.

In general, B is represented in JAVA by a method, and can also be viewed as M_{k+1}. Inversely, methods $M_j ... M_k$ $(j \geq 1)$ can be seen as part of the comparison. In that sense, we provide several shortcuts for common methods, e.g., to compare the type of an object to a predefined one. This reflects the method `isInstance()` (the dynamic counterpart to the `instanceof` operator) in `java.lang.Class`.
In our condition classes, like the `Equals` class given in Figure 8 (representing an equality test in the sense of II), we have added constructors which alleviate their use. The third constructor in the figure for instance enables the expression of nested method calls by providing a URL-like string denoting the names of the methods. The accessor is in that case created implicitly. Figure 9 shows the links between our JAVA implementation of accessors and conditions, illustrated through the `Equals` and `Invoke` classes.

```
public final class Equals
  implements Condition, java.io.Serializable
{
  /* compare the message object as a whole */
  public Equals(Object to) {...}
  /* compare return value of accessor */
  public Equals(Accessor acc, Object to) {...}
  /* implicit accessor creation */
  public Equals(String names, Object[][] args,
                Object to) {...}
  ...
  public boolean conforms(Object m) {...}
}
```

Figure 8: `Equals` Class (Excerpt)

[10] The counterpart to the well-known `Magnitude` type in SMALLTALK [GR83].

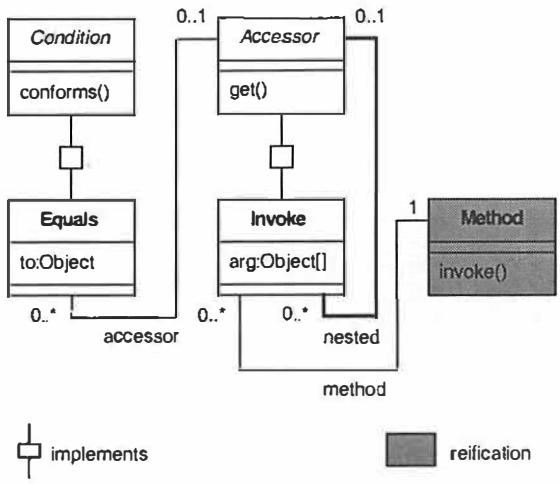


Figure 9: Class Diagram

3.4 Subscription Patterns

A subscription pattern S represents a combination of basic conditions.

Patterns and conditions. A subscription pattern $S = ((C_1, ..., C_n), F)$ is characterized by a set of n basic conditions, which are all evaluated for a given message m, and a n-ary function $F(b_1 : bool, ..., b_n : bool) \dashrightarrow bool$, which is evaluated for the results of these conditions: $S(m) = F(C_1(m), ..., C_n(m))$. A pattern is thus evaluated like a condition: $S(o : obj) \dashrightarrow bool$, and is represented in JAVA by an object of type `Condition`. The model diverges here from the concrete realization, in that the function F does not appear as such.

Expressing patterns. F is namely explicitly constructed by *combining* conditions. These combinations are expressed through specific conditions, reflecting binary predicates, like `And` (Figure 10), `Or`, etc. Furthermore, we propose a condition `Not` for negation. To ease the expression of combinations, we introduce the `SimpleCondition` interface (Figure 11), an extension of `Condition`, which our basic conditions in fact implement.[11]

This subscription scheme based on conditions inherently expresses the subscription grammar. Syntax

[11] This counteracts JAVA's lack for operator overloading (as provided for instance by C++ [Str97]).

errors known from subscription languages, where they are only recognized at execution of the parser, are here detected by the JAVA compiler.

```
public final class And
  implements Condition, java.io.Serializable
{
  /* the two arguments */
  private Condition first;
  private Condition second;

  public And(Condition first,
             Condition second)
    { this.first = first; this.second = second; }

  public boolean conforms(Object m)
    { return first.conforms(m) &&
        second.conforms(m); }
  ...
}
```

Figure 10: And Class (Excerpt)

```
public interface SimpleCondition
  extends Condition

{
  public SimpleCondition and(Condition with);
  public SimpleCondition or(Condition with);
  public SimpleCondition nand(Condition with);
  public SimpleCondition nor(Condition with);
  public SimpleCondition xor(Condition with);
  public SimpleCondition not();
}
```

Figure 11: `SimpleCondition` Interface

4 Programming Example

This section illustrates the use of content-based filters through *chat sessions* based on simple DACs. We first recall our notion of DISTRIBUTED ASYNCHRONOUS COLLECTION (DAC) and then build on the example initially introduced in [EGS00a].

4.1 Background: Distributed Asynchronous Collections

Just like a conventional collection, a DAC represents a group of related objects. A DAC is however distributed and can be accessed from various nodes of a network. In a way similar to a JAVASPACE, a DAC enables distributed participants to share information by pulling information from the space, but also by registering a callback object to be notified of future elements. DACs furthermore express through their type the *qualities of service* (QoS) they support. In other terms, we offer a framework of DAC subtypes representing different semantics and QoS. In this example, we will use a DAC representing a topic, to which application components subscribe with an optional content-based filter.

4.2 A Chat Scenario

We concentrate on two chat addicts, Alice and Bob, who love to chat deep into the night. Therefore they subscribe to the topic */Chat/Insomnia* to receive all messages from like-minded chatters. Figure 12 shows class `ChatMsg`, which represents a possible message class for this application.

```
public class ChatMsg
  implements java.io.Serializable
{
  private String sender;
  private String text;
  public String getSender() { return sender; }
  public String getText() { return text; }
  public ChatMsg(String sender, String text) {
    this.sender = sender; this.text = text; }
}
```

Figure 12: Event Class for Chat Example

4.3 Publishing

A DAC represents a topic, and in order to access such a DAC from a process, a proxy must be created. This requires an argument denoting the name of the topic represented by the DAC. Except for that topic name, the action of creating a DAC proxy is identical to creating a local collection. The `DAStrongSet` collection class instantiated in Figure 13 offers reliable delivery of notifications to subscribers. The instance called `myChat` henceforth provides access to the topic */Chat/Insomnia*. Now it is possible to directly publish and receive messages for the topic associated to that DAC.
Creating an event notification for a topic consists in

inserting a message object into the DAC by issuing a call to the `add()` method, from where it becomes accessible for any party.

```
...
/* connect */
DASet myChat = new DAStrongSet("/Chat/Insomnia");

/* create new message and publish it */
ChatMsg m = new ChatMsg("Alice", "Hi everyone");
myChat.add(m);
...
```

Figure 13: Publishing a Message

4.4 Subscribing

In order to subscribe to a DAC, a callback object implementing the `Notifiable` interface must be provided. Figure 14 shows how to implement a simple callback object for chat sessions.

```
public interface Notifiable {

  public void notify(Object m);
}

public class ChatNotifiable
  implements Notifiable
{
  public void notify(Object m) {
    /* elements are of type ChatMsg */
    ChatMsg cm = (ChatMsg)m;
    System.out.println("Message from " +
                       cm.getSender());
    System.out.println(cm.getText());
  }
}
```

Figure 14: Callback Object

Episode I. Figure 15 shows a first example of content-based publish/subscribe. We suppose here that Bob is only interested in what a particular participant, Alice, publishes on topic */Chat/Insomnia*. Bob defines a corresponding condition. The `null` argument in the condition initialization denotes a set of empty argument lists. A subscription is viewed as an interest in future elements, and is expressed by a call to the `contains()` method.

```
...
/* connect */
DASet myChat = new DAStrongSet("/Chat/Insomnia");

/* create condition */
Condition onlyAlice =
  new Equals("/getSender", null, "Alice");

/* create callback object and subscribe */
Notifiable n = new ChatNotifiable();
myChat.contains(n, onlyAlice);
...
```

Figure 15: Content-Based Subscribing (I)

Episode II. Suppose now that Bob is only more interested in what Alice says about him. For this second condition, the text carried by each chat message must be checked for the occurrence of Bob's name. Remember that in JAVA a string `s1` can be checked for the occurrence of a substring `s2` by asking `s1` through a call to `indexOf()` for the index of its first occurrence of `s2`. If `s2` is not contained in `s1`, the call returns `-1`. The resulting second condition in the figure represents all messages which do *not* contain Bob's name, and must therefore be negated.
This example shown in Figure 16 illustrates how to easily combine basic conditions with the `SimpleCondition` interface, and how the application can specify the type of the message objects with implicit accessor creation, as required for the performance optimizations we propose in the next section.

5 Performance

Reflective systems and meta-level architectures offer increased modularity and flexibility. The benefit of such dynamism is often, but not necessarily, diminished by performance degradation. In this section we first give a rough idea of the cost of dynamic code introduced by JAVA reflection. Motivated by these results we then propose two optimizations to our system, and we discuss their performances.

5.1 Preliminary: Cost of Reflection

According to the way we have described our implementation, methods are invoked dynamically, i.e.,

```
...
/* connect */
DASet myChat = new DAStrongSet("/Chat/Insomnia");

/* create first condition, with type specification */
SimpleCondition onlyAlice =
  new Equals("ChatMsg:/getSender", null, "Alice");

/* create args list and corresp. second condition */
Object[][] args = {{null}, {"Bob"}};
SimpleCondition notAboutMe =
  new Equals("ChatMsg:/getText/indexOf", args,
             new Integer(-1));

/* combine conditions */
SimpleCondition pattern =
  onlyAlice.and(notAboutMe.not());

/* create callback object and subscribe */
Notifiable n = new ChatNotifiable();
myChat.contains(n, pattern);
...
```

Figure 16: Content-Based Subscribing (II)

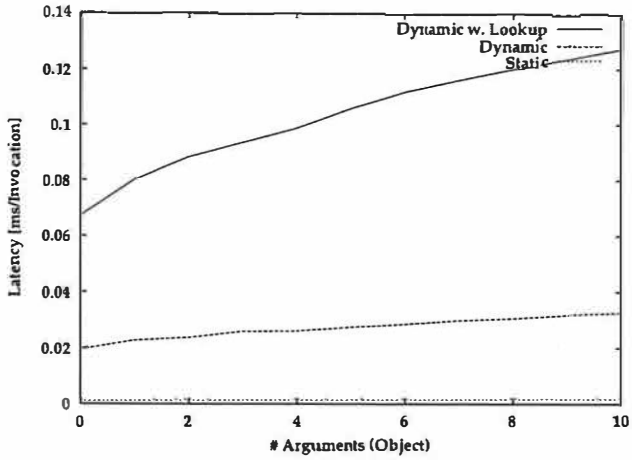


Figure 17: Latency with Different Invocation Styles

through reified methods. Such dynamic invocations are much more expensive than static ones. Moreover, when subscribing to structurally conformant objects (cf. Section 3), method objects are obtained at runtime for each message object. Such lookups are very costly, and are summed with the overhead of the dynamic invocations.

Figure 17 shows the cost of dynamic calls by comparing the overhead of local method invocations with a varying number of arguments (between 0 and 10 objects). These are performed using (1) dynamic invocations, each combined with a method lookup, (2) dynamic invocations without lookups, and (3) static invocations. These tests were made on a SUN ULTRA 60 (SOLARIS 2.6, 256 Mb RAM, 9 Gb harddisk) with JAVA 1.2 (native threads). The test setting did not involve any JUST IN TIME (JIT) compiler. The speedup factor observed for static invocations when using a JIT compiler was over three. The speedup in the case of dynamic evaluation is, as expected, insignificant.

5.2 Optimizations

The amount of expensive dynamic code can be reduced if the type of the message objects is known. The type information can be given to the system either (1) by using reflection explicitly, or (2) by specifying the type of the message objects by name. The type information enables the application of optimizations.[12]

Avoiding redundant invocations. Message objects are usually matched against patterns of several subscribers at a time, and these patterns are likely to present redundancies. We discuss here an optimization based on that observation, which is similar, but not identical to the *tree matching* algorithm used in GRYPHON [ASS+98]. The tree matching algorithm factors out redundant subpatterns with simplified assumptions: only *and*s of basic conditions are considered, and latter ones are primitive comparisons of attribute values with predefined values.

In contrast, our filter library offers more expressiveness, e.g., nested method invocations, different comparators and combinations (*and*, *or*, ...). Such combinations are performed statically, and dynamic queries on message objects represent the critical factor in our system. As a consequence, we focus on detecting common denominators of accessors, in order to avoid the evaluation of redundant dynamic method invocation chains. Figure 18 shows a simple example of redundant accessors where each subscriber specifies a pattern consisting of a single basic condition. An *invocation tree*, like the one shown in Figure 19, is constructed from all accessors and is evaluated for every filtered message object.

[12] In a companion paper [EGS00b] we introduce *type-based* publish/subscribe: a static classification scheme based on the *types* of message objects. The *type-based* publish/subscribe scheme ensures type safety, and thus enforces optimizations through the inherent knowledge of types and makes type

Subscriber S_1: $A_{1_1} = ((M_1, P_1))$
Subscriber S_2: $A_{2_1} = ((M_2, P_2), (M_3, P_3))$
Subscriber S_3: $A_{3_1} = ((M_2, P_2), (M_3, P_3), (M_4, P_4))$
Subscriber S_4: $A_{4_1} = ((M_2, P_2), (M_3, P_5))$

Figure 18: Redundancy between Accessors

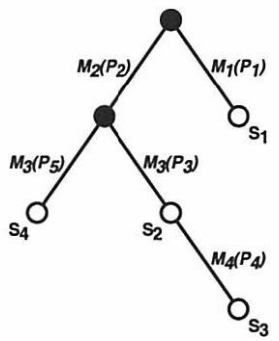


Figure 19: Invocation Tree

Enforcing static filters. Based on the observation that dynamic invocations are far more costly than static ones, we have implemented an alternative optimization. Any dynamic invocations are avoided by generating static source code from accessors after performing type checks. The source code is then directly compiled by calling the SUN JAVA compiler (`sun.tools.javac`), in a way similar than this is done in [KMS98] or [TCKI00].[13]

5.3 Evaluation

We evaluate here the benefits of the two above optimizations by comparing the resulting performances with two non-optimized scenarios. These are namely (1) the filtering of structurally conformant messages, and (2) the filtering of type conformant messages.

Testbed. Our measurements were made with the JAVA VM 1.2, enabled JIT and native threads on SUN SOLARIS 2.6. A single producer was publishing message objects encapsulating a single string from one network (SUN ULTRA 60, 256 Mb RAM, 9 Gb harddisk), to subscribers equally distributed over two further networks; one composed of all together 60 SUN SUPERSPARC 20 stations (model 502: 2 CPU, 64 Mb RAM, 1Gb harddisk), and the second one composed of 60 SUN ULTRA 10 (256 Mb RAM, 9 Gb harddisk) stations. The individual stations and the different networks where communicating via FAST ETHERNET.

Parameters. We have made a set of extensive tests, in which we have always varied one of four parameters for the subscriptions. These are namely, (1) the fraction of positive matches for a subscriber $1/c$, (2) the total number of subscribers s, (3) the maximum nesting level of invocations for queries a, and (4) the number of different query methods d at each nesting level.

Varying 1/c*:* From n produced messages, an average of n/c messages matched a given subscribers pattern. Figure 20(a) shows the effect of varying c. It confirms the intuition that the cost of sending messages with UDP does not depend on the matching scheme, and can be seen as fixed. With $c > 100$ in this scenario, the pure cost of matching is measured. In order to accentuate the differences between the matching schemes without contradicting our concrete applications, we have chosen $c = 10$ for the next figures.

Varying s: Similarly to the scenario in Figure 18, we have chosen one basic condition per subscriber. Figure 20(b) reports the effect of scaling up s, conveying that the two optimizations are almost equivalent with a large s.[14] As shown in the previous figure, UDP is a limiting factor with an increasing number of sends (here due to a large s). Performance drops faster with static filters, since every additional subscriber involves a full pattern evaluation. In contrast, the optimized dynamic scheme is less sensitive since redundant queries are avoided.

Varying a: The probability of having $i \in [0, a]$ nested invocations was chosen as $p_a = 1/(a+1)$. Increasing a reduces throughput with static invocations (Figure 21(a)), since static accessors comprise more invocations. Similarly, the optimized

checks and casts inside the application code superfluous.

[13] [KMS98] terms this technique (runtime) *linguistic* reflection, which is seen as a synonym of *structural* reflection.

[14] Our system relies on a hierarchical topology of message brokers, among which membership information is split up. A single process rarely has knowledge of more than 100 participants.

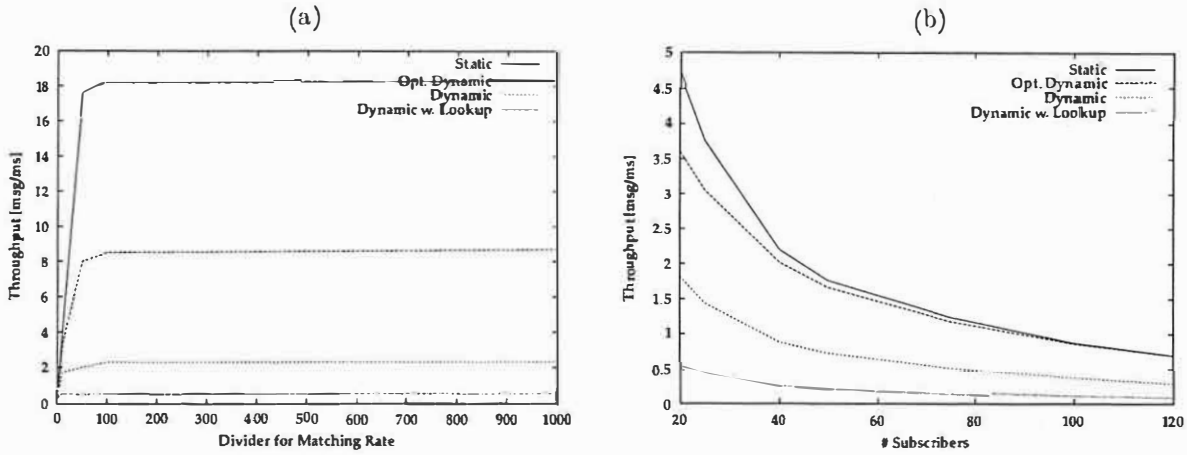


Figure 20: Matching Rate and Number of Subscriptions

dynamic scheme is less efficient with an increasing a, since the total number of performed methods increases with the depth of the tree.

Varying d: One of d methods was chosen at each nesting level with a probability of $p_d = 1/d$. Varying d obviously does not influence static filter evaluation. On the other hand, increasing d might lead to increasing the potential number of edges leaving from any node in the invocation tree. The resulting performance loss is directly visible in Figure 21(b). The optimized dynamic scheme is however more penalized by increasing a, as shown in the previous figure. This is due to the fact that increasing a by 1 might result in up to d new edges in every former leaf of the invocation tree.

Interestingly the optimized dynamic matching scheme never overperforms the static scheme, even if the speedups become close with a large number of message sends. One could believe that with a strong redundancy between patterns and a large number of subscribers the dynamic scheme would become more efficient. Even with extreme parameter values, we have however never encountered such a scenario.

6 Discussion

In this section we debate alternative models and realizations we have considered as potential solutions for an adequate content-based subscription scheme in the context of DISTRIBUTED ASYNCHRONOUS COLLECTIONS.

Application-defined filters. We promote the expression of subscription patterns as a combination of instances of our predefined condition classes. An alternative to this consists in allowing the application to provide directly its own static filter objects (byte code). Patterns expressed this way are however opaque and not necessarily correct nor safe, and make optimizations difficult.
Nevertheless, we have opted for an open design, i.e., separation of interfaces and classes (e.g., `Accessor/Invoke`) vs conditions and accessors as `final` classes. This enables the extension of our subscription API with application-defined accessors and conditions. Our proposed optimizations can still be enforced by following certain design guidelines.

Towards a unified language. An alternative to our subscription API consists in using the JAVA language itself as the subscription language. That is, providing code in a stringified form (source code), that can be parsed and compiled at runtime. A pseudo-variable `m` would represent the runtime message object, and method invocations could be directly expressed, e.g.:

```
"m.getSender().equals("Alice") && ...;"
```

The evaluation of the code given here as a string is *deferred*. This comes to introducing two levels of programming, in a way similar to [NN88]. The generalization of that approach leads to *multi-stage*

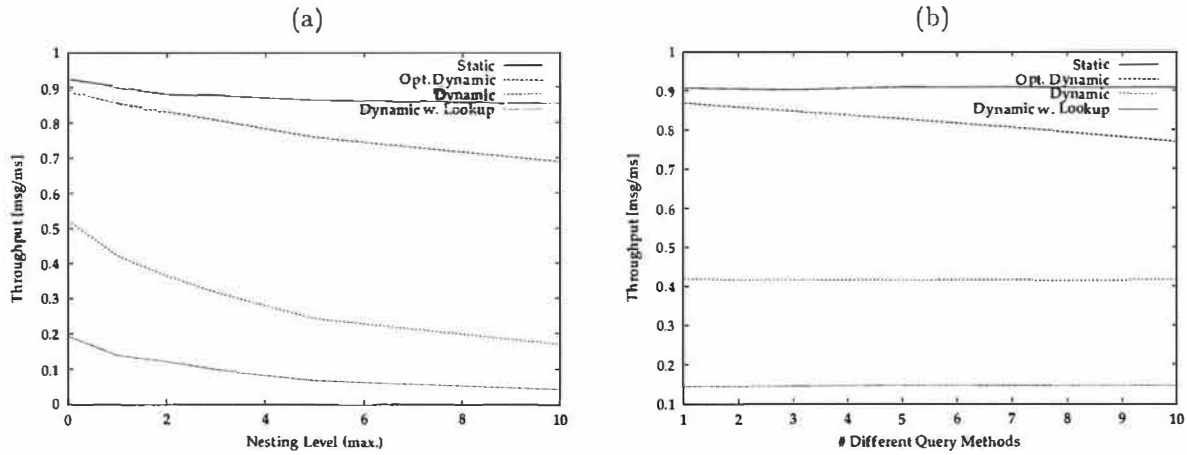


Figure 21: Expressiveness and Redundancy of Subscriptions

programming, e.g., METAML [TS97]. METAML is a meta-programming language that was designed as a homogenous runtime code generator toolkit. In METAML, the evaluation of expressions in <> (called brackets) is deferred to the next stage, in a sense similar to our stringified code delimited by "" above. Expressions evaluated at a later stage can refer to constructs at a previous stage. When stringifying meta-code in JAVA as above, this is not possible, since JAVA reflection does not allow to dynamically obtain a reference to a variable by its name. This limitation can be circumvented as long as invocation arguments can be constructed inside the pattern string (e.g., `"Alice"`), but poses problems for complex matches. Extending the JAVA language in the sense of METAML would have contradicted our resolution of using merely standard language constructs.

Java reflection. JAVASSIST [Chi00] and OPENJAVA [TCKI00] are two approaches to extending JAVA with load-time structural reflection, i.e., the ability of *modifying* classes at runtime prior to instantiation. OPENJAVA promotes a compile-time meta-object protocol [Chi95] based on an extension of `java.lang.Class`, and makes use of the SUN JAVA compiler, while JAVASSIST provides an extended `Classloader` supporting the creation of new methods as copies of existing ones.
We have however refrained from using JAVASSIST or OPENJAVA, because our static filters represent very specific classes which can be generated without any language extension.

7 Concluding Remarks

We argue through our work that, unlike what is often claimed (e.g., [Koe99]), message-oriented middleware and object-oriented principles are not contradictory. In [EGS00a], we have made a first step, by introducing a programming abstraction called DISTRIBUTED ASYNCHRONOUS COLLECTION (DAC) which is versatile enough to express commonalities between the different message-oriented interactions styles. In that paper we have focused on topic-based publish/subscribe.
In this paper, we have attacked another bastion, content-based publish/subscribe, which is presumed to contradict object-oriented principles by its very nature. We have illustrated that it is indeed possible to express content-based subscription patterns in a way that fully preserves encapsulation. Moreover, we have shown that our approach offers further practical benefits over contemporary approaches, like the possibility to prevent syntax errors and type errors.
In terms of performance, the cost of our solution is incurred by the latency resulting from the *use* of JAVA reflection. In our case, this use is however reduced to verifications of subscription patterns aiming at avoiding type errors. After this initialization phase, dynamic invocations are circumvented by using static code generated at runtime without any modification to the JAVA compiler or VM. The throughput of our system is thus not conditioned by the use of reflection, as proven by the resulting performances. We are furthermore currently working on a new optimization scheme combining the benefits of our static and dynamic optimizations. The

idea is to generate static code from dynamic invocation trees, to further improve performance but also to reduce the overall compilation effort.
The cost of our solution in terms of feasibility is limited to the *need* for structural reflection; yet with such minimal features that the inherent JAVA reflection capabilities can satisfy this need.
We do not claim that our content-based subscription scheme is the ultimate solution to content-based publish/subscribe, nor that it replaces existing specifications. It should rather be seen as a pragmatic attempt to circumventing shortcomings of other approaches. Our filter library is not limited to the context of DACs, but could be put to work easily in other existing event-based systems.

Acknowledgments

We would like to thank both Andrew Black and Joe Sventek for their valuable comments. Those comments have helped us concretize the ideas presented in this paper.

References

[AEM99] M. Altherr, M. Erzberger, and S. Maffeis. iBus - a software bus middleware for the Java platform. In *International Workshop on Reliable Middleware Systems of the 13th IEEE Symposium On Reliable Distributed Systems (SRDS'99)*, pages 43–53, October 1999.

[ASS+98] M.K. Aguilera, R.E. Strom, D.C. Sturman, M. Astley, and T.D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC'99)*, November 1998.

[BCM+99] G. Banavar, T. Chandra, B. Muhkerjes, J. Nagarajarao, R.E. Strom, and D.C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, 1999.

[BHL95] B. Blakeley, H. Harris, and J.R.T. Lewis. *Messaging and Queuing Using the MQI: Concepts and Analysis, Design and Development.* McGraw-Hill, 1995.

[BW98] M. Biichi and W. Weck. Compound types for Java. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, pages 362–373, October 1998.

[BZ87] T. Bloom and S.B. Zdonik. Issues in the design of object-oriented database programming languages. In *Proceedings of the 2nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, pages 441–451, 1987.

[Car98] A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks.* PhD thesis, Politecnico di Milano, December 1998.

[Chi95] S. Chiba. A metaobject protocol for C++. In *Proceedings of the 10th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, pages 285–299, October 1995.

[Chi00] S. Chiba. Loadtime structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, pages 313–336, June 2000.

[Coi87] P. Cointe. Metaclasses are first class: The ObjVlisp model. In *Proceedings of the 2nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, pages 156–167, October 1987.

[Cor99] Talarian Corporation. *Everything You need to know about Middleware: Mission-Critical Interprocess Communication (White Paper).* http://www.talarian.com/, 1999.

[DEC94] DEC. *DECMessageQ: Introduction to Message Queuing*, April 1994.

[EGS00a] P.T. Eugster, R. Guerraoui, and J. Sventek. Distributed Asynchronous Collections: Abstractions for publish/subscribe interaction. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, pages 252–276, June 2000.

[EGS00b] P.T. Eugster, R. Guerraoui, and J. Sventek. Type-based publish/subscribe. Technical Report DSC/2000/029, Swiss Federal Institute of Technology, Lausanne, http://dscwww.epfl.ch/EN/publications/, June 2000.

[Fer89] J. Ferber. Computational reflection in class based object-oriented languages. In *Proceedings of the 4th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*, pages 317–326, October 1989.

[FHA99] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice.* Addison-Wesley, June 1999.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, January 1985.

[GR83] A.J. Goldberg and A.D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[HBS98] M. Happner, R. Burridge, and R. Sharma. Java Message Service. Technical report, Sun Microsystems Inc., October 1998.

[KMS98] G. Kirby, R. Morrison, and D. Stemple. Linguistic reflection in java. *Software - Practice and Experience*, 28(10):1045–1077, 1998.

[Koe99] P. Koenig. Messages vs. objects for application integration. *Distributed Computing*, 2(3):44–45, April 1999.

[Mic97] Microsoft. *Microsoft Message Queuing Services*, 1997.

[NN88] F. Nielson and H.R. Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, January 1988.

[OAA+00] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman. Exploiting IP Multicast in content-based publish-subscribe systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2000)*, pages 185–207, April 2000.

[Obe00] R.J. Oberg. *Understanding & Programming COM+.* Prentice Hall, 2000.

[Obj99] ObjectSpace. *JGL - Generic Collection Library.* http://www.objectspace.com/jgl/, 1999.

[OMG98] OMG. *CORBAservices: Common Object Services Specification, Chapter 4: Event Service.* OMG, December 1998.

[OMG00] OMG. *Notification Service Standalone Document.* OMG, June 2000.

[OPSS93] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus - an architecture for extensible distributed systems. In *14th ACM Symposium on Operating System Principles*, pages 58–68, December 1993.

[PO93] R.J. Peters and M.T. Özsu. Reflection in a uniform behavioral object model. In *Proceedings of the 12th International Conference on Entity-Relationship Approach*, pages 37–49, December 1993.

[Pow96] D. Powell. Group communications. *Communications of the ACM*, 39(4):50–97, April 1996.

[RW97] D. Rosenblum and A. Wolf. A design framework for internet-scale event observation and notification. In *6th European Software Engineering Conference/ACM SIGSOFT 5th Symposium on the Foundations of Software Engineering*, pages 344–360, September 1997.

[SA97] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of the Australian UNIX and Open Systems User Group Conference (AUUG'97)*, http://www.dtsc.edu.au/, September 1997.

[SBS98] D.C. Sturman, G. Banavar, and R. Strom. Reflection in the Gryphon message brokering system. In *Reflection Workshop of the 13th ACM Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, 1998.

[Ske98] D. Skeen. *Vitria's Publish-Subscribe Architecture: Publish-Subscribe Overview.* http://www.vitria.com, 1998.

[SO95] D.D. Straube and M.T. Özsu. Query optimization and execution plan generation in object-oriented data management systems. *IEEE Transactions on Knowledge and Data Engineering*, 7(2), April 1995.

[Str97] B. Stroustrup. *The C++ Programming Language, Third Edition.* Addison-Wesley, 1997.

[Sun99a] Sun. *Java Core Reflection API and Specification*, 1999.

[Sun99b] Sun. JavaSpaces specification. Technical report, Sun Microsystems Inc., November 1999.

[Sun99c] Sun. Jini Entry specification. Technical report, Sun Microsystems Inc., November 1999.

[SV97] D. Schmidt and S. Vinoski. Overcoming drawbacks in the OMG Event Service. *SIGS C++ Report magazine*, 19(6), June 1997.

[Sys00] BEA Systems. *Reliable Queuing Using BEA Tuxedo: White Paper.* http://www.beasys.com/products/, 2000.

[TCKI00] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. *A Class-based Macro System for Java*, pages 119–135, LNCS 1826. Springer-Verlag, July 2000.

[TIB99] TIBCO. *TIB/Rendezvous White Paper.* http://www.rv.tibco.com/, 1999.

[TS97] W. Taha and T. Sheard. Multi-stage programming. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 321–321, June 1997.

PSTL—A C++ Persistent Standard Template Library*

Thomas Gschwind†
tom@infosys.tuwien.ac.at
http://www.infosys.tuwien.ac.at/Staff/tom/

Institut für Informationssysteme
Technische Universität Wien
Argentinerstraße 8/E1841
A-1040 Wien, Austria

Abstract

The C++ Standard Template Library provides efficient storage of data in containers, and efficient operations on such containers. While STL can be parameterized with custom allocators, these cannot be used to add persistency to the container classes provided by STL. Thus, we have designed the Persistent Standard Template Library (PSTL) that overcomes this by providing its own containers that are compatible with STL, but store their elements on disk. This compatibility provides a programming model that is known and more natural to C++ programmers and enables the reuse of many of the algorithms provided by STL in combination with PSTL. In this paper we discuss PSTL's design, show the challenges we faced, and how STL's design would have to be extended to provide native support for persistency.

1 Introduction

Persistent container libraries such as [GND99, Sle00] have two key advantages. Compared to volatile containers, the size of their database can grow beyond the size of the available memory and compared to transactional databases, they exhibit less overhead since they do not provide any functionality to rollback transactions. Persistent container libraries are the key element of many programs such as sendmail [CA97], `NNTPCache` [AB], or NewsCache [GH99].

Today, a large number of persistent storage libraries exists, ranging from simple text-files to complex databases. The most prominent among these are the various types of `dbm` databases used for instance by sendmail. The creators of the Java programming language have even added a type to Java to indicate whether a given object may be stored on external storage [AG97]. Any object implementing the `Serializable` interface can be serialized and deserialized transparently.

We have not found any persistent container for C++ that is compatible with STL and fulfills the requirements to be used within a server application. The importance of persistent libraries has already been identified by Bjarne Stroustrup. According to [Ste95], his assumption was that persistency could be provided using custom allocators. This assumption, however, was based on an early version of STL that had allocators that were real objects [Str94, Str00]. In later versions of STL the allocator mechanism was simplified due to logical and performance problems with that kind of generality. However, it is still unclear whether these problems would have been unsurmountable [Str00]. Problems with the current allocator mechanism have also been identified by [Ste98b] and will be explained in detail in Section 3.

We think that compatibility with STL is one of the key requirements since it allows for a more natural object-oriented programming style and the reuse of many of the algorithms provided by STL. These algorithms range from iterating over the container's elements to sorting the container and exchanging elements between different containers. Compatibil-

*This work was supported in part by a grant from the USENIX Advanced Computing Association.

†Part of this work was implemented during an internship at Hewlett Packard Labs, Palo Alto, CA 94304.

ity is also one of the key challenges since some key properties of persistent containers such as serialization are not necessary for their volatile counterparts.

The paper is structured as follows. Section 2 gives a brief description of the requirements for a persistent container library. Section 3 explains why the allocator mechanism provided by STL is not sufficient to fulfill these requirements. The design and implementation of PSTL is presented in Section 4 along with the challenges we faced in preserving compatibility with STL. Section 5 presents an evaluation and related work is considered in Section 6. We outline our ideas for improving PSTL in Section 7 and draw our conclusions in Section 8.

2 Requirements

The requirements for persistent containers are simple. Data has to be stored on persistent storage, each container should use its own file, and the container should be able to store more elements then fit into the computer's memory (main memory plus swap space). Additionally, the containers should be simple to use and compatible with existing standards.

Since data is stored on disk, the objects stored in the container need to be serialized. Depending on the object, a `memcpy()` operation might be sufficient. In the following, we refer to these objects as *simple objects* and to those that require special serialization functionality (i.e., objects using pointers) as *fragile objects*.

Since persistent containers are frequently used by server daemons to store their databases (e.g., sendmail, NewsCache, ...), the database must be accessible by several different processes. This is because daemons handle several clients simultaneously and typically spawn a new process for each client connection. Depending on the daemon, the database is created once and accessed read-only or is manipulated during the program's operation. If the container is manipulated, support for locking is needed and changes need to be visible to other processes immediately.

In case of the C programming language, these requirements are fulfilled by [GND99], for instance. For C++, however, we could not find a persistent container class library fulfilling these requirements and seamlessly integrating with STL.

PSTL was primarily developed to replace NewsCache's [GH99] newsgroup and article database. Thus, the containers have to be efficient enough for handling the typical Usenet traffic. The typical spool size of a Usenet News server is about 60GB and the traffic is at least 3–5GB of article data per day [CBB97] excluding the traffic generated by news reading clients. To ease the maintenance of the spool area, NewsCache stores each newsgroup in a separate file/container.

3 STL and Persistence

The containers provided by STL are stored in main memory which is volatile, but can be parameterized with different memory allocators. Thus, our initial approach was to provide a custom allocator that allocates its elements on persistent storage.

While the constructors of the container classes do not directly provide an argument to pass the name of the file the container should be stored in, they provide constructors taking a custom allocator class as argument that could encapsulate the container's filename.

The custom allocator class, however, only provides methods that allow to allocate and free memory and to construct and destroy a given object [Str97, ISO98]. There is no mechanism to query the allocator for previously stored elements. Thus, the constructors of STL's container classes do not check for pre-existing elements. The only way to fix this problem is to subclass the original STL class and replace the constructors and destructor as identified in [Ste98b, Ste98a].

It is insufficient, however, to simply replace the constructors and destructor of the original STL container except in the simple case where the constructor reads all elements from the file and the destructor writes all elements back to the file [Ste98b]. This is due to the fact that typical STL implementations make certain but legitimate optimizations based on the internal representation of the container. For instance, GNU C++'s STL implementation uses a pointer as the vector's iterator. While this is fine for a non-persistent container, it cannot be used for a persistent container as we will show in the following sections.

Another drawback is that a standard allocator is assumed to hold no per-object data [Str97, Chap-

ter 19.4.3] allowing the library to implement some container-manipulation functions by relinking elements. For instance, `splice()` may be implemented by moving an element from one list to another without copying the element. This is not possible if different lists can be stored in different files. In this case the elements have to be copied from one container/file to the other. Thus, the allocator is bound to the type the container is parameterized with rather than to the container. Hence, this would restrict a program to use the same file for all containers parameterized with the same value type.

Due to these reasons it is not possible to convert an STL container into a persistent STL container just by supplying a different allocator class to its constructor. Adding persistence requires a tighter coupling between the container and its corresponding allocator. Thus, we were forced to reimplement the containers provided by STL. The containers provided by PSTL are compatible with their corresponding STL containers, but use an extended interface provided by our persistent allocators.

4 Design and Implementation

Since PSTL tries to adhere to the STL specification, most of PSTL's design is equivalent to STL's design. The differences between PSTL and the typical STL implementation are outlined in the following sections. For instance, we had to add a serialization mechanism that provides transparent serialization of the container's elements. Additionally, we added an argument to the container's constructors to indicate the file where the elements should be stored in. This allows the user to instantiate a container without having to explicitly instantiate the corresponding persistent allocator.

4.1 Serialization

As mentioned in Section 2 special care needs to be taken when *fragile objects* need to be stored in the persistent container. C/C++ pointers cannot be stored because the data stored at the address the pointer is pointing to will likely be located at a different address at the program's next invocation. This problem can be solved by using a pointer swizzling technique as presented in [SKW92] or by serializing the object before storing it on disk. Since the pointer swizzling technique has a major drawback with regards to concurrent accesses, as we will point out in Section 6.5, we have chosen to store the objects in a serialized form.

STL itself does not provide any functionality for the serialization of its elements because the container's elements are stored in memory and do not have to be made persistent. Thus, we had to extend the interfaces defined by STL with a means for serializing and deserializing fragile objects. Typically, this functionality is implemented using one of the following approaches.

- Instrumentation of the data structures [Kni99].
- Requiring the user to provide the code for serializing and deserializing the data structures stored within the container [GND99, Sle00].

The first approach allows the container to identify the type of the data stored at each position. This makes it possible to use a generic algorithm for garbage collection and defragmentation of the database. The price for this, however, is an increased size of the container and that only instrumented data structures may be stored. Another disadvantage is that the layout of the data structure is pre-determined and cannot be changed at run-time thus making the exploitation of polymorphism difficult. Since we want our implementation to be as compatible with the STL as possible, and no garbage collection mechanisms are provided by STL anyway, we have chosen the second approach.

A straightforward implementation would be to require the classes of the data structures to be stored to provide a member function that implements the serialization and deserialization functionality. This implementation, however, would have two shortcomings. It cannot be easily added to any existing class and it does not work in combination with builtin types.

Thus, we have decided to use traits [Mye95, Str97] for PSTL. Traits allow us to extend a template argument without requiring it to provide the extension. The extension is moved to a trait class which is supplied as an additional template argument to the container class. An advantage of this implementation is the possibility to use different serialization algorithms for different container instances parameterized with the same value type.

```
template <class T, class Tr=serialize_trait<T>>
class pvector {
  // typedefs, ...
  reference
  front() {
    // See Section 4.5 for a description
    // of getdata
    offset_type head=alloc.getdata();
    return Tr::deserialize(head);
  }
}
```

Figure 1: Vector Using Serialization Trait

A simplified version of the pvector container using the serialization trait class is shown in Figure 1. Whenever the container needs to serialize or deserialize an object it calls the corresponding functions of the trait class. The container and the serialization classes have both a reference (alloc) to the persistent allocator class and can use its functions for allocating memory on the persistent file and converting offsets to pointers and vice versa.

Our persistent container classes provide trait classes for storing builtin types, objects that do not have any pointers, and strings. The traits for the serialization of other classes must be provided by the user of the persistent template library. The implementation of a custom serialization trait is fairly simple. The interface that has to be provided is shown in Figure 2.

```
template <class T> struct serialize_trait {
  // typedefs, ...
  pstl_serialize_traits(allocator_type &a)
    : alloc(a) {}
  void
  serialize(const T &t, offset_type o) {
    // serialize t to alloc.off2ptr(o)
  }
  reference
  deserialize(offset_type o) {
    // deserialize element and return a
    // reference to it
  }
  const_reference
  deserialize(offset_type o) const {
    // deserialize element and return a
    // constant reference to it
  }
  size_type
  size() {
    // return T's size; in case of a complex
    // object use an offset here and allocate
    // extra memory via the persistent
    // allocator
  }
}
```

Figure 2: Sample Serialization Trait

4.2 Low-Level Disk Access

So far, we have not discussed how and when the data is stored onto disk. Typical solutions to this problem are:

- Read the data from disk in the constructor of the container and write it back in its destructor [Ste98b]. Unfortunately, this approach violates two of our requirements. It does not allow the container to grow beyond the memory size and modifications of the container won't be visible to other processes until the container is freed or flushed explicitly.
- Whenever an element needs to be accessed, seek to the element's position and read it from or write it to disk [GND99, Nel98]. This requires the elements to be read from disk and to be copied from/to the I/O buffers every time an element is accessed. Since all modern operating systems provide elaborate caching strategies, the disk access is negligible. The real problem is that every element needs to be copied at each access no matter whether it actually would have to be serialized or not.

PSTL, however, uses memory mapped files, an approach different from both of the approaches presented above. Memory mapped files have the advantage that the operating system maps the file associated with the container into the program's address space and the program can read and write the file like memory allocated using `malloc()`.

As shown in Figure 3, PSTL maps the container's file storing the serialized representation of the container into its address space. Whenever an object is accessed, it is deserialized and thus copied. In the Figure below, this is the case with the container's first element, the author's address. Now, the program can interact with the object representing the element like with any other object. Finally, when the object is destroyed, it is serialized back to the memory mapped file.

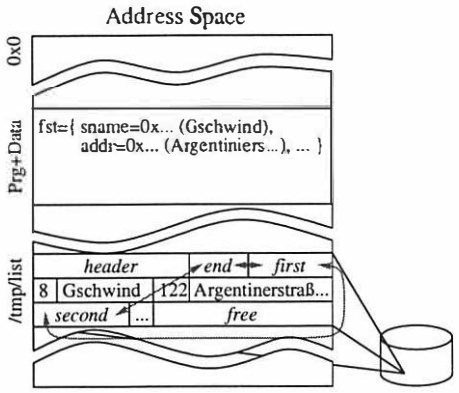


Figure 3: Serialization and Deserialization of Elements

Using memory mapped files has several advantages:

- The program can operate directly on the memory mapped file. No copying from and to I/O buffers is necessary.
- When virtual memory gets tight, the contents of the container's associated file are swapped back to the file instead of to the swap space.
- User programs may use pointers pointing into the memory mapped file and may act on them like on all other pointers.
- *Simple objects* do not have to be deserialized and copied from the container. Since their deserialized representation is the same as their serialized one, it is sufficient to return a reference/pointer to the object's address in the memory-mapped area.

While this approach sounds simple, much care needs to be taken with its implementation. The size of the area to be mapped into memory needs to be known a priori. If more space is required than originally requested, the file needs to be resized and the memory area needs to be remapped. In this case all pointer references to the memory mapped region might have to be calculated anew since it cannot be guaranteed that the resized container can be remapped to the same address as before [Bac86, LMKQ90].

Theoretically, it would be possible to map different parts of the file to different memory regions and thus getting around this problem. This, however, would require the container to maintain a mapping for each region and address calculation would get complicated and most probably inefficient.

Unfortunately, no memory management is available for the memory provided by the memory mapped file. Thus, we had to implement our own persistent allocator class that manages the free blocks of the memory mapped file. The interface provided by the persistent allocators is explained in Section 4.5. Similar to STL, this allocator class is used by all persistent containers.

4.3 References

The most challenging part of PSTL with regard to STL-compatibility was the references returned by some of the container's member-functions (e.g., `front()`). For this problem, however, it is important to distinguish between fragile objects (objects using pointers) and simple objects (objects without pointers).

If the container only stores simple objects, it is sufficient to return a reference to the memory address the object is stored at. This is because the object's serialized and deserialized representation is the same and the object's size is well known and does not change. Hence, there is no danger of accidentally overwriting the container's internal data.

If the container stores fragile objects, however, the object needs to be deserialized and thus copied into a temporary buffer. Otherwise, the user would not be able to use it. However, if the deserialized copy changes, we want the serialized version of the element to be changed as well.

The straightforward approach would be the provision of a wrapper class overriding all of the original methods and serializing the element back to disk when necessary. This is ugly, however, and is not guaranteed to work since some of the class's non-virtual functions might have been expanded inline. Fortunately, this problem can be solved cleanly using templates.

PSTL returns a reference object that encapsulates the position of the data structure in the memory mapped area, provides a deserialized copy of the data structure and in case of a non-constant reference writes the element back to disk when the reference object is destructed. This is achieved by a generic wrapper similar to the one shown in Figure 4. The wrapper only needs to be parameterized with the value-type's serialization trait.

```
template <class Tr>
class __pstl_ref: public Tr::value_type {
  // typedefs, ...
  serializer_type *s;
  offset_type o;
public:
  __pstl_ref(const value_type &x,
             Tr *ser, offset_type off)
    : value_type(x), s(ser), o(off) {}
  ~__pstl_ref() {
    s->destruct(o);
    s->serialize(*(value_type*)this,o);
  }
};
```

Figure 4: PSTL's Reference Class

While PSTL's reference objects ensure that changes will be written back to disk after the element's modification, the element will be written back to disk only after the reference's destruction. To prevent multiple wrappers of the same element from interfering with each other a cache object should keep track of the instantiated wrapper classes and ensure that no more than one wrapper is instantiated for a given element within one process. The locking mechanism ensures that different processes are only allowed to have constant references at the same time.

4.4 Locking

Since we want several processes to be able to access PSTL's container's simultaneously, we added a locking mechanism that ensures mutual exclusion. The current implementation of PSTL simply adds two new functions to each container: `lock()` and `unlock()`. `lock()` locks the container for shared or exclusive access and `unlock()` unlocks the container. PSTL also maintains an internal lock stack to ensure that different functions locking the same container do not interfere with each other. For instance, a function might request to lock a container that has already been locked by its process (i.e., by its caller). In this case `lock()` does not have to lock the container since it is already locked. When the function unlocks the container, however, the container must not be unlocked since the container should be still locked by the caller.

Based on the design of STL, however, it should be possible to add a transparent locking facility. Whenever an iterator is requested, the container could be locked and with the iterator's destruction it could be unlocked. Depending on a constant or non-constant iterator, the container would be locked shared or exclusively. The same would apply to references returned by the container. Unfortunately, depending on the container's value type PSTL uses a wrapper class or a builtin C++ reference. While the wrapper class would support a transparent locking mechanism, the latter does not since C++'s builtin references do not provide a destructor that could be used for unlocking the container. A simple solution of course would be to always return a wrapper object. This issue will be attacked in future versions of PSTL.

4.5 Implementing More Persistent Containers

In case the container types provided by PSTL are not sufficient, it is easy to implement new ones that better fit the requirements of special purpose applications. The implementation of a persistent container is similar to the implementation of a volatile container, except that offsets have to be used instead of pointers and instead of allocating memory using `new`, it has to be requested from the persistent allocator class. The persistent allocator class provides the following functions for the management of the container's persistent memory:

`off2ptr()/ptr2off()` convert offsets to pointers and vice versa. While offsets must be stored in the container, they need to be frequently converted to pointers to operate on the memory-mapped storage area.

`nvalloc()/nvfree()` allocates/frees a memory area within the file. Whenever one of these functions is called, the container's storage area might have to be resized and might have be remapped to a different memory area. Thus, pointers into the persistent storage area need to be recalculated or verified after calling this function.

`getdata()/setdata()` returns/sets the offset to the *root* memory area of the container's data structure. Using `getdata()`, the persistent containers gain access to pre-existing elements at the container's construction time.

5 Evaluation

We evaluated PSTL in terms of compatibility by adapting some sample applications we had written previously to be used by the containers provided by PSTL instead of those provided by STL. Though PSTL has not yet been tuned for performance, we will also give a short performance evaluation to give the reader a rough estimate of PSTL's current performance.

5.1 Compatibility

The main difference from STL is the extension to allow the user to supply a custom serialization functionality. While we have included some serialization classes for common value types, it is likely that the user will have to supply a custom serialization trait for his own value types. Additionally, the filename of the container needs to be specified when instantiating a PSTL container.

```
vector<int> v();                    // STL vector
pvector<int> pv("/tmp/vector"); // PSTL vector
pvector<int,pallocator,myserializer>
    pv2("/tmp/vector2");    // use my serializer
```

When converting our test applications from using the containers provided by STL to the ones provided by PSTL, we identified that users typically make assumptions about the implementation of the containers. A common assumption is that the iterator of a vector is implemented as a pointer or that a reference is implemented as a C++ reference. While this works in combination with most STL implementations, it is not compliant with the standard [ISO98].

```
T* i=pv.begin();                        // error
pvector<T>::iterator j=pv.begin();      // ok
T& r=pv.front();                        // error
pvector<T>::reference s=pv.front();     // ok
```

As we have explained in Section 4.3, PSTL uses its own wrapper class as a reference. Fortunately, this is identified by the compiler and thus can be corrected by the user easily. The same applies to references returned by PSTL.

PSTL's non-constant references copy their elements back to disk, regardless whether they have been modified or not. This is not the case for constant references as they must not be changed by definition and thus do not have to be written back to disk. Hence, if performance is of importance, one should distinguish between member functions returning constant (e.g., `front() const`) and non-constant references (e.g., `front()()`).

C++ resolves the member function to be called based on its function name and the arguments including the implicit `this` argument. Only in case of a constant object pointer/reference, the constant method will be chosen. The following code clarifies this.

```
pvector<int> pv1("/tmp/filename");
const pvector<int> &pv2=pv1;


/**** ok, but inefficient ****/
pvector<int>::reference ref=
  pv1.front();  // call ref begin()
pvector<int>::const_reference cref1=
  pv1.front();  // call ref begin() and
                // convert ref to const_ref


/**** ok, and efficient ****/
pvector<int>::const_reference cref2=
  pv2.front();  // call const_ref begin() const
```

A set of polymorphic elements is typically stored by parameterizing the container with the pointer to the base class (`<base_type*>`) and allocating the elements on the heap. Achieving the same behavior with PSTL requires the user to specify a custom serialization trait for `base_type*`. Serialization is straightforward by calling the appropriate serialization function. To allow for extensibility the deserialization function should be implemented using exemplar types [Cop92] (sometimes also referred to as virtual constructors).

If the performance of manipulating persistent elements is of major concern the user may specify his own reference type within the serialization trait. This allows the reference type to gain access to PSTL's data allocator and manipulate the data structure in the persistent storage area directly.

Even though, in case of concurrent access to the containers, PSTL requires to lock and unlock the containers explicitly, this is not a compatibility problem. STL implementations do not implement persistent containers and thus do not allow simultaneous access. In case a program should be retrofited to allow multiple processes to access the same data

structure, it is necessary to review the code for possible deadlock situations anyway. In future versions of PSTL, however, we will try to implement a transparent locking facility as mentioned in Section 4.4.

5.2 Performance

So far, our efforts on PSTL focused on compatibility with STL and not on its performance. However, we were still curious to see how it would perform in comparison to Berkeley DB [Sle00] (with logging for recovery or transactions disabled) and gdbm [GND99], the leading persistent container libraries available for Unix.

The computer used for this benchmark was a Pentium II (350MHz), 256MB of RAM, and a Seagate 4GB hard drive (ST34323A). The computer was running RedHat 6.1 (Linux kernel 2.2.12, glibc 2.1) and all container libraries and test applications were compiled using gcc-2.95.2 using -O2 for optimization. Each application was executed 5 times and the median was chosen for our performance comparison.

Due to the lack of available benchmarks for the evaluation of persistent container libraries, we have implemented our own applications: an address book mapping family names to addresses and phone numbers and a resource reservation system. Both applications are based on associative containers with the difference that the resource reservation system uses a simple object as key and thus favoring PSTL. In the following, however, we will limit our discussion to the address book application (Table 1). The results of the resource reservation system are available from the PSTL web site together with the source code of the benchmarks.

	59840 entries			148397 entries	
Database	PSTL	BDB	gdbm	PSTL	BDB
Insertion	9.23	14.73	15.59	31.65	45.28
Iteration	4.30	7.32	8.50	10.30	20.28
Lookup	2.21	5.10	3.00	5.86	16.00
Deletion	69.23	16.27	12.53	482.19	27.58

Table 1: Address Book Benchmark (seconds)

Table 1 shows the results for two different database sizes. One with 59840 entries without duplicates since gdbm does not support duplicate keys and one with 148397 entries with duplicates. *Insertion* refers to inserting all the elements, *iteration* to iterating over all of the elements 10 times, *lookup* looking up each element, and *deletion* to removing all elements one after the other.

Astonishingly, in many cases PSTL performs better than its competitors. We assume that one of the reasons is PSTL's use of memory mapped files.

Typically, Berkeley DB does not use memory mapped files since this would restrict the size of the database to the size of the address space. With the advent of 64bit computers, however, we do not believe this to be a problem for PSTL. Berkeley DB only uses memory mapped files for databases opened read only and smaller than a given threshold. If the smaller Berkeley DB database is opened read only, iterating over the elements takes 6.43 seconds and 2.03 seconds for looking them up. This does not show much potential for performance improvement of PSTL here. It is interesting to note that Berkeley DB scales better for inserting new elements and PSTL scales better for iterating over elements and looking them up. This might be due to the fact that PSTL uses a red-black tree and Berkeley DB a B-tree.

GNU gdbm uses a hash table for its internal representation. This also gets clear by looking at its fast lookup speed, even though it uses normal disk I/O. Iteration over the elements is poor since gdbm only returns the key when iterating over the container requiring another lookup for the associated value. If the key is sufficient, the time for iterating over the elements is just 4.05 seconds.

PSTL's performance, however, for deleting the elements shows plenty of potential for improvement. This is due to the fact that PSTL uses a linear list for the management of free blocks sorted by their location within the file. Deleting all of the elements one after the other in random order populates the list with a huge number of elements even though adjacent free blocks are merged immediately. This problem will be attacked in future versions.

6 Related Work

Currently, several persistent container class libraries are available. The most prominent are the various `*dbm` databases.

6.1 dbm and Variants

dbm is one of the oldest libraries that provide access to persistent containers. Under Unix, dbm [Unia] and its newer variants (ndbm [Unib], gdbm [GND99], Berkeley DB [Sle00]) are a de-facto standard for persistent information storage.

The most advanced of the dbm databases is Berkeley DB. Unlike the older variants, it does not only provide a hash table for its internal representation, but also a B-tree and two different kinds of queue formats as well as optional transaction management. Additionally, it provides bindings for C, C++, Java, and Tcl.

Berkeley DB requires the elements to be stored in the database to be serialized before the database allocates the element's memory forcing the element to be copied twice. Additionally, the serialization function does not have access to Berkeley's memory allocator and is forced to store the whole element in one chunk. On the other hand, this approach ensures that the database cannot be easily corrupted by the serialization algorithm.

Even though Berkeley DB provides C++ bindings, it provides a very low level API like the older dbm variants. Most of its functions work on DBTs (Data Base Thangs) representing raw regions of memory. Whenever a key/value pair needs to be stored in a database, both key and value need to be converted into a DBT. Additionally, when a value corresponding to a key needs to be requested, the key needs to be converted into a DBT and the database returns a DBT representing the value.

PSTL, however, has an API compatible with STL and the serialization function is part of the container's type. Besides that, the serialization function is called transparently and the user does not have to deal with it. Based on our work on PSTL it might be fairly simple to provide a C++ API for Berkeley DB which is compatible with STL too.

6.2 Disk Based Container

The disk containers presented in [Nel98] use the same approach as used by the *dbm databases to access elements stored in its containers. Similar to PSTL, the elements in the database are referenced by offsets instead of pointers.

Unfortunately, this work seems to be more an experiment on how persistency could be achieved in C++ without going into much detail. For instance, the management of the free blocks in the container is overly simple and only provides a fixed-size block allocation scheme. This makes it difficult and inefficient to use these containers in combination with variable sized elements. The library is also incompatible with the containers provided by STL.

6.3 Persistent Template Library

The Persistent Template Library is presented in [Ste98b] and [Ste98a]. PTL is a library that provides containers compatible with the ones provided by STL. Unfortunately, the containers have some severe drawbacks:

- Whenever a PTL container is allocated it is copied from disk into main memory. Afterwards it behaves like the STL containers. This is trivial since it uses the same code. When the container is destroyed, its data is written back to disk.
- Due to the above, the size of the container is limited to the size of the main memory. A persistent container, however, should be able to accommodate more elements than fit into the memory. This demand is essential for programs that have to manage huge amounts of data.
- Only one process may instantiate a container at one time since the container is copied into main memory and other processes will see the changes only when it is saved back to disk at destruction.

This work is interesting because the author comes to the conclusion that it is not possible to use the allocator mechanism provided by STL to add persistency. The author has solved the problem by subclassing the corresponding container provided by the STL and by supplying his own constructors for reading the elements from disk and destructor for writing them back. This, unfortunately, led to the above disadvantages.

6.4 POST++

While POST++ does not provide persistent containers directly, it provides a simple and effective storage for application objects [Kni99] and supports the use of different storages for different objects.

Except for a slight instrumentation of the data structures to be stored persistently, POST++ transparently manages the persistence of the objects. For the instrumentation of the data structures, POST++ uses C preprocessor macros that register the attributes to be made persistent. A special macro is provided for the identification of pointers as their management is more complex. Due to the explicit identification of pointers, POST++ even provides garbage collection to reclaim unused storage.

POST++ uses a different but nevertheless interesting approach. The choice whether to use PSTL or POST++ depends largely on the application domain. For instances, POST++ does not provide persistent containers per se, it only provides the infrastructure to make your objects (including container objects) persistent. Thus, if special purpose data structures need to be made persistent and their instrumentation is possible, POST++ might be a good choice. Otherwise, we recommend the use of PSTL.

6.5 Texas

Texas [SKW92] is a persistent storage system similar to POST++ but instead of requiring the user to instrument the data structures, it uses a pointer swizzling technique in combination with runtime type descriptors and slightly modified heap allocation routines. Runtime type descriptors are generated using an optional feature of gcc.

Objects are either allocated on the conventional (transient) heap, or the persistent heap. In the implementation presented in [SKW92], all the objects allocated on the persistent heap are stored within a single file, but the authors claim that it would not be difficult to remove this restriction.

While Texas provides a simple and powerful way to manage data structures located on persistent storage, it has some shortcomings with regards to sharing the persistent database. If several different processes need to share the same database, they need to share the same persistent page mappings. While this might be simple in case of a single file used for all persistent objects, it cannot be easily achieved if different files are used for different persistent objects and the files to be shared between the processes (or the processes sharing the files) cannot be known a priori.

7 Future Work

For the maintenance of free blocks, PSTL uses a linear list, sorted by the memory location of the free blocks. This representation is highly efficient for debugging purposes since it allows an efficient way to check the allocation status of the whole file and whether the free list has been corrupted (e.g., by a method writing past its allocated memory area). As we have shown in Section 5.2, however, it is inefficient from a performance point of view. To alleviate this problem, we plan to provide a better allocator class using a binary tree for the management of free blocks.

PSTL does not yet implement all the containers efficiently. It might be interesting to see whether PSTL could profit from a data structure optimized for block-sized access patterns like a B or B+tree [Com79] or whether the operating system's cache management is sufficient.

We also plan to provide an associative container optimized for lookups using a hash table. It might also be interesting to see whether gdbm's algorithms can be reused for this container to achieve not only interoperability with STL, but also with the most widely used type of persistent container.

In future versions of PSTL we will also try to implement a transparent locking mechanism as explained in Section 4.4.

8 Conclusions

Persistent STL (PSTL) containers provide a variety of benefits not available with existing persistent container implementations available for C++. They provide a variety of different persistent containers, allowing the container's size to grow beyond the available memory, and supporting STL's object-oriented programming model known to many C++ programmers.

In this paper we have shown the problems of adding persistency to STL and demonstrated why it is impossible to use STL's allocator mechanism to add persistency to existing STL containers. This impossibility forced us to implement our own container classes that are interface compatible with STL. Based on this implementation, we have pre-

sented how STL's design had to be extended to support persistency.

Based on PSTL, we explained how to implement a transparent serialization facility without breaking interface compatibility with the corresponding STL containers and without requiring any support from the objects to be stored in the persistent container. We solved this using a traits approach that encapsulates the serialization functionality and allows different containers to use different serialization algorithms.

Except for the declaration of the container, elements are serialized and deserialized transparently. This has been solved using a reference wrapper that works in combination with both objects using virtual and non-virtual member functions. Additionally, the persistent allocator class used by PSTL's containers provides a simple but efficient means to create new persistent container classes for special purpose applications.

In an evaluation of the library we identified that STL containers can easily be replaced with PSTL containers. Only little modifications are necessary in case of implicit assumptions about the container's implementation. This, however, is detected by the C++ compiler. We have also included a performance evaluation which will serve as a basis for future improvements of PSTL.

Acknowledgements

This paper could never have been accomplished without the help of several people. I would like to thank Pankaj Garg, my supervisor during my internship with HP labs, for his support of the project, Mehdi Jazayeri, my supervisor at Technische Universität Wien, for his continuous support, and my colleague Georg Trausmuth for his suggestion to add the serialization functionality using traits. Finally, I would like to thank Bjarne Stroustrup for shepherding this paper.

Availability

PSTL is distributed under the GNU Public License and is available at http://www.infosys.tuwien.ac.at/-NewsCache/pstl.html.

References

[AB] Julian Assange and Luke Bowker. NNTPCache. http://www.nntpcache.org/.

[AG97] Ken Arnold and James Gosling. *The Java Programming Language (Java Series)*. Addison-Wesley, 2nd edition, December 1997.

[Bac86] Maurice J. Bach. *The Design of the Unix Operating System*. Prentice-Hall, 1986.

[CA97] Bryan Costales and Eric Allman. *sendmail*. O'Reilly, January 1997.

[CBB97] Nick Christenson, David Beckemeyer, and Trent Baker. A Scalable News Architecture on a Single Spool. *;login:*, 22(3):41–45, June 1997.

[Com79] Douglas Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[Cop92] James O. Coplien. *C++ Programming Styles and Idioms*. Addison-Wesley, August 1992.

[GH99] Thomas Gschwind and Manfred Hauswirth. NewsCache—A High Performance Cache Implementation for Usenet News. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 213–224. USENIX, June 1999.

[GND99] Pierre Gaumond, Philip A. Nelson, and Jason Downs. *GNU dbm: A Database Manager*, 1.5 edition, 1999. For GNU dbm, Version 1.8.

[ISO98] ISO/IEC. *ISO/IEC14882: Programming Languages—C++*, 1st edition, July 1998.

[Kni99] Konstantin Knizhnik. Persistent Object Storage for C++. http://-www.ispras.ru/~knizhnik/post.html, March 1999.

[Lip98] Stanley B. Lippman, editor. *C++ Gems*. Cambridge University Press, 1998.

[LMKQ90] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1990.

[Mye95] Nathan C. Myers. Traits: A New and Useful Template Technique. *C++ Report*, June 1995.

[Nel98] Tom Nelson. Disk-Based Container Objects. *C/C++ Users Journal*, pages 45–53, April 1998.

[SKW92] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: An Efficient, Portable Persistent Store. In *Proceedings of the Fifth International Workshop on Persistent Object Systems*, September 1992.

[Sle00] Sleepycat Software. The Berkeley Database 3.1.17, July 2000. http://-www.sleepycat.com/.

[Ste95] Al Stevens. Al Stevens Interviews Alex Stepanov. *Dr. Dobb's*, March 1995. http://-www.sgi.com/Technology/STL/drdobbs-interview.html.

[Ste98a] Al Stevens. AntiPatterns and PTL 2. *Dr. Dobb's*, pages 114–116, April 1998.

[Ste98b] Al Stevens. The Persistent Template Library. *Dr. Dobb's*, pages 117–120, March 1998.

[Str94] Bjarne Stroustrup. Making a Vector Fit for a Standard. *The C++ Report*, 6(8), October 1994. Also in [Lip98].

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[Str00] Bjarne Stroustrup. Personal Email Communication, October 2000.

[Unia] University of California, Berkeley. *dbm(3) Unix Programmer's Manual*.

[Unib] University of California, Berkeley. *ndbm(3) 4.3BSD Unix Programmer's Manual*.

Making Java Applications Mobile or Persistent

Sara Bouchenak*

SIRAC Laboratory (INPG-INRIA-UJF)
INRIA Rhône-Alpes, 655 av. de l'Europe, 38330 Montbonnot St Martin, France.
Sara.Bouchenak@inria.fr
** INPG (Institut National Polytechnique de Grenoble)*

Abstract

Today, mobility and persistence are important aspects of distributed applications. They have many fields of use such as load balancing, fault tolerance and dynamic reconfiguration of applications. In this context, the Java virtual machine provides many useful services such as dynamic class loading and object serialization which allow Java code and objects to be mobile or persistent. However, Java does not provide any service for the mobility or the persistence of control flows (threads), the execution state of a Java program remains inaccessible.

We designed and implemented new services that make Java threads mobile or persistent. With these services, a running Java thread can, at an arbitrary state of its execution, migrate to a remote machine or be checkpointed on disk for a possible subsequent recovery.

Therefore migrating a Java thread is simply performed by the call of our *go* primitive, by the thread itself or by an external thread. In other words, the migration or the checkpointing of a thread can be initiated by the thread itself or by another thread.

We integrated these services into the JVM, so they provide reasonable and competitive performance figures without inducing an overhead on JVM performance. Finally, we experimented a dynamic reconfiguration tool based on our mobility service and applied to a running distributed application.

Keywords: mobility, persistence, migration, checkpointing, recovery, Java, thread, JVM

1. Introduction

Today, mobility and persistence are important aspects of distributed applications and have several fields of use [Milojicic99] [Ambler99]. Application mobility can be used to dynamically balance the load between several machines in a distributed system [Nichols87], to reduce network traffic by moving clients closer to servers [Douglis92], to dynamically reconfigure distributed applications [Hofmeister93], to implement mobile agent platforms [Chess95] or as a machine administration tool [Oueichek96]. Application persistence can be used for fault tolerance [Wojcik95] or for application debugging.

Distributed applications development is an important research direction in computing systems. In this context, the object paradigm has proven to be well suited to distributed applications development and the Java Virtual Machine (JVM) is now considered as a reference platform [Gosling96]. The Java compiler produces *bytecode*, an intermediate code that is interpreted by the JVM. Today, the JVM is ported on almost every platform and can therefore be viewed as a universal machine.

In order to facilitate the development of distributed applications, the JVM provides several services [Sun00a] among which:

- Object serialization. The serialization service allows the transfer of Java objects between several nodes or the storage of objects on disk.
- Dynamic class loading. The dynamic class loading service enables the transfer of Java code between several nodes.

Therefore, Java provides useful services for the mobility and the persistence of code and data. However, Java does not provide any service enabling the mobility and the persistence of applications during their execution. Thus, if a running Java application migrates to a new location, only using object serialization and dynamic class loading, the execution state of the application is lost. In other words, when arriving on its new location, the migratory application can access to its code and its re-actualized data but it has to restart the execution from the beginning. Consequently, the provided Java services are not sufficient for either enabling dynamic load balancing of distributed Java executions or allowing the state of running applications to be checkpointed and then recovered.

We designed and implemented new services that make Java threads, i.e. executions, mobile or persistent. With these services, a running Java thread can, at an

arbitrary state of its execution, migrate to a remote machine or be checkpointed on disk for a possible subsequent recovery.

Our *java.lang.threadpack* Java package provides several primitives, among which *go* performs thread migration, *store* is used for thread checkpointing and *load* for thread recovery. Therefore migrating a Java thread is simply performed by the call of the *go* primitive, by the thread itself or by an external thread. In other words, the migration or the checkpointing of a thread can be initiated by the thread itself or by another thread.

We integrated these services into the JVM, so they provide acceptable performance figures without inducing an overhead on JVM performance. Finally, we experimented with a prototype implementation a dynamic reconfiguration tool based on our mobility service and applied to a running distributed application.

The rest of this paper consists of three main parts. We first describe our service for capturing/restoring Java thread state in section 2 and then present the services of thread mobility and thread persistence in section 3. In sections 4 and 5, we respectively present performance figures and describe some experiments that we performed with our services. Finally, we discuss related work and present our conclusions and future directions in section 6 and 7.

2. Thread state capture/restoration service

Both services allowing the mobility and the persistence of Java threads are based on a common service: a thread state capture/restoration service. We first describe the representation of a thread state in the JVM and then present the design principles of our capture/restoration service and its implementation details.

2.1. Java thread state

The JVM can support the concurrent execution of several threads [Lindholm96]. The state of a Java thread is illustrated by figure 1, it consists of three main data structures:

- *The Java stack*. A Java stack is associated with each thread in the JVM. The Java stack consists of a succession of *frames* (see figure 2). A new frame is pushed onto the stack each time a Java method is invoked and popped from the stack when the method returns. A frame includes the local variables of the associated method and the partial results of this method. The values of local variables and partial results may be of several types: integer, float, Java reference, etc. A frame also contains registers such as the program counter and the top of the stack.
- *The object heap*. The heap of the JVM includes all the Java objects created during the lifetime of the JVM. The heap associated with a thread consists of all the objects used by the thread (objects referenced in the thread's Java stack).
- *The method area*. The method area of the JVM includes all classes (and their methods) that have been loaded by the JVM. The method area associated with a thread contains the classes used by the thread (classes whose some methods are references by the thread's Java stack).

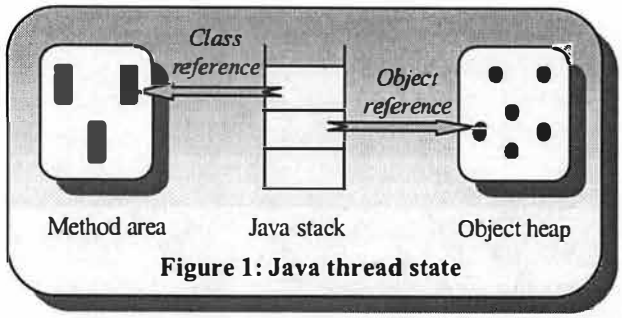


Figure 1: Java thread state

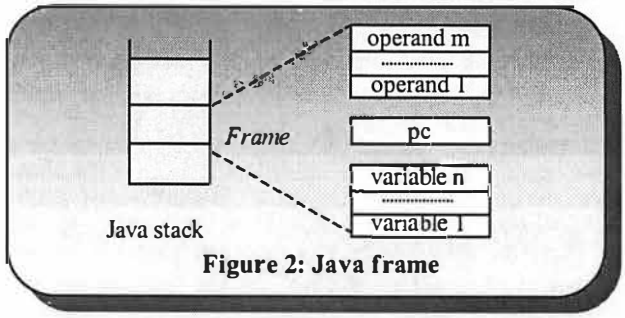


Figure 2: Java frame

2.2. Design of the capture/restoration service

Here are the design principles and the design decisions of our Java thread state capture/restoration service.

2.2.1. Design principles

The thread state capture/restoration service enables, on the one hand, the *capture* of the current state of a running thread, and on the other hand, the *restoration* of a previously captured state in a new thread: the new thread starts running at the point at which the execution of the previous thread was interrupted.

Thread state capture consists in interrupting the thread during its execution and extracting its current state. The extraction amounts to build a data structure (a Java object) containing all information necessary for restoring the Java stack, the heap and the method area associated with the thread. To build such a data

structure, the Java stack associated with the thread is scanned in order to identify the objects and the classes that are referenced from the stack. After state capture, the resulting data structure can be serialized and sent to another virtual machine in order to implement mobility or it can be stored on disk for persistence purpose.

One of our motivations was to provide a generic service which allows the implementation of various capture policies. Consequently we rely on Java object serialization and dynamic class loading features in order to capture respectively the heap and the method area.

The restoration of a thread consists first in creating a new thread and initializing its state with a previously captured state. After that, the Java stack, the heap and the method area associated with the new thread are identical to those associated with the thread whose state was previously captured. Finally, the new thread is started, it resumes the execution of the previous thread.

2.2.2. Design decisions

There were mainly two problems for designing a Java thread state capture/restoration service. The first issue is to have access to the state of Java threads, a state that is internal to the JVM. The second issue is that the state of Java threads is not portable on heterogeneous architectures.

2.2.2.1. Non accessible state

The state of Java threads is internal to the JVM. This state is not accessible by Java programs and can therefore not be directly captured. In order to allow the capture of threads state, we extended the JVM and externalized the state of Java threads.

2.2.2.2. Non portable state

Unlike the heap and the method area that consist of information portable on heterogeneous architectures (Java objects and Java classes), the Java stack is a native data structure (C structure). The representation of the information contained in the Java stack depends on the underlying architecture. The thread state capture service must translate this non portable data structure (C structure) to a portable data structure (Java object).

Translating the Java stack into a portable data structure consists more precisely in translating the native values of local variables and partial results (figure 2) into Java values. This translation requires the knowledge of the types of the values. But the Java stack does not provide any information about the types of the values it contains: a four bytes word may represent a Java reference as well as an *int* value or a *float* value. The thread state capture service must recognize the types of the values contained in the Java stack. We propose two approaches for type recognition:

- *Type recognition at runtime.* The first approach consists in recognizing the types during runtime. The type information is built in parallel with thread execution. The type information is actualized each time a bytecode instruction is interpreted by the thread because the bytecode instructions are typed and are applied to particular types [Lindholm96]. Therefore, at state capture time, the type information is available. The drawback of this approach is that it induces an overhead on application performance.
- *Type recognition at capture time.* The second approach consists in recognizing the types at capture time. The type information is built by analyzing the bytecode to determine the execution path of the thread. This analyze is similar to the bytecode verifier algorithm [Lindholm96]. This approach avoids any overhead on application performance but causes a latency due to type information building.

We first implemented the approach based on type recognition at runtime [Bouchenak00] and then implemented the approach based on type recognition at capture time*. In this paper, we focus our attention on the design principles of our services and do not tackle the implementation details.

2.3. Implementation of the capture/restoration service

Our Java thread state capture/restoration service was integrated to the Java 2 SDK (formerly called JDK 1.2) [Sun00a]. Our new Java package, called *java.lang.threadpack*, provides many classes such as the *ThreadState* class whose instances represent the state of Java threads and the *ThreadStateManagement* class that provides the necessary features for capturing and restoring Java threads state.

Figure 3 illustrates a part of the application programming interface (API) of the *ThreadStateManagement* class. The *capture* method allows the capture of the current state of a Java thread, the captured state is returned as a result of this method, as a *ThreadState* object. Symmetrically, the *restore* method creates a new Java thread, initializes its state with the *ThreadState* argument, starts the new thread and returns it as a result of the method. The new thread resumes the execution of the thread whose state was

* The evaluation presented in section 4 concerns the first approach

previously captured and passed as an argument of the *restore* method.

java.lang.threadpack
Class ThreadStateManagement
public final class **ThreadStateManagement** extends Object
The ThreadStateManagement class provides several useful services for the capture and restoration of Java thread states.

Method Summary	
static ThreadState	capture(Thread thread) Captures the state of the thread argument and returns it as a ThreadState object.
static Thread	restore(ThreadState threadState) Creates a new Java thread, initializes it with a previously captured state and starts its execution.
static void	captureAndSend(Thread thread, SendInterface sndItf, boolean toStop) Captures the state of the thread argument and sends it (to a remote node or to the disk) by calling the sendState method of the SendInterface interface.
static Thread	receiveAndRestore(ReceiveInterface rcvItf) Receives the state of a Java thread by calling the receiveState method of the ReceiveInterface interface, creates a new Java thread, initializes it with the received state and starts its execution.

Figure 3: Interface of the capture/restoration service

The *captureAndSend* and *receiveAndRestore* methods are generic and can specialize the capture and restoration operations to application needs. Besides capturing the state of a Java thread, the *captureAndSend* method allows the programmer to specify the way the captured state is handled: the captured state can for example be sent to a remote machine for a mobility purpose, it can be stored on disk in the case of application persistence, etc. The specialization of the handling of the captured state is specified by the second argument of the *captureAndSend* method. In fact, this argument implements our *SendInterface* interface and so provides a *sendState* method that is called by our *captureAndSend* method, just after the capture of the thread state. The third argument of the *captureAndSend* method is a boolean that specifies if the thread whose state is captured is stopped or resumed. This argument is for example set to *true* in the case of thread migration and is set to *false* for remote thread cloning.

Symmetrically, the *receiveAndRestore* method specifies the way a thread state is received before it is restored: the state can for example be received from a remote machine, it can be read from disk, etc. The specialization of the way the thread state is received is possible thanks to the argument of the *receiveAndRestore* method: this argument implements our *ReceiveInterface* interface and so provides a *receiveState* method that is called by our *receiveAndRestore* method, just before the restoration operation.

3. Thread mobility and thread persistence services

Besides our system service for capturing/restoring the state of Java threads, we provide higher-level services for the mobility and the persistence of Java applications.

Making an application mobile is the action of moving an application, during its execution, from one node to another: the application starts running, on the new node, at the point at which the execution was interrupted on the first node. Therefore, making a Java application mobile consists in making the underlying Java thread mobile. In the case of a multi-threaded application, the whole group of threads has to be moved.

Making a thread mobile is the action of capturing the current state of the thread, sending this state to a target machine and restoring the state in a new thread on the target machine: the new thread resumes the execution in the state left by the original thread.

In the same way, application persistence consists first in saving the current state of a running application on stable storage (disk). The saved state can subsequently be restored in order to resume the execution of the application. Therefore, making a Java application persistent consists in making the underlying Java thread(s) persistent.

Making a thread persistent is, first, the action of capturing the current state of the thread and saving it on disk and then, the ability of restoring the saved state in a new thread: the new thread resumes the execution of the previous thread.

Our *MobileThreadManagement* class belongs to the *java.lang.threadpack* package and provides services necessary for the mobility of Java threads. Figure 4.a illustrates a part of the application programming

interface of this class. The *go* method allows the transfer of a running Java thread to a Java virtual machine identified by an IP address and a port number. And the *arrive* method enables the reception of a Java thread coming from a machine specified by an IP address and a port number.

java.lang.threadpack
Class MobileThreadManagement
public final class **MobileThreadManagement** extends Object
The MobileThreadManagement class provides several useful services for making Java threads mobile.

Method Summary	
static void	go(Thread thread, String tagetHost, int targetPort) Moves the execution of the thread argument to the machine specified by the host name and the port number arguments.
static Thread	arrive(String sourceHost, int sourcePort) Receives a thread from the machine specified by the host name and the port number arguments.

Figure 4.a: Service for the mobility of Java threads

```
public static void go(Thread thread, String targetHost,
                      int targetPort) {

MySender sndItf = new MySender(targetHost,
targetPort) ;
ThreadStateManagement.captureAndSend(thread,
sndItf) ;

}
```

```
public static Thread arrive(String sourceHost,
                            int sourcePort) {

MyReceiver rcvItf = new MyReceiver(sourceHost,
                                   sourcePort) ;
return
ThreadStateManagement.receiveAndRestore(rcvItf) ;

}
```

Figure 4.b: Implementation of mobility service

The *go* and *arrive* methods are implemented using respectively the *captureAndSend* and *receiveAndRestore* generic methods (see section 2.3). The *go* method is implemented as follows:

- The *go* method calls the *captureAndSend* method (figure 4.b).
- The *captureAndSend* method is adapted using an instance of the *MySender* class.
- The *MySender* class implements the *SendStateInterface* interface and therefore provides a *sendState* method, this method aims at establishing a connection to a machine and sending the *ThreadState* object using serialization (figure 4.c).

```
class MySender
   implements SendInterface {

String host ;
int port ;

MySender(String host, int port) {
   this.host = host ;
   this.port = port ;
}

public void sendState(ThreadState state) {
// Send state to <host, port>.
...
}
}
```

```
class MyReceiver
   implements ReceiveInterface {

String host ;
int port ;

MyReceiver(String host, int port) {
   this.host = host ;
   this.port = port ;
}

public ThreadState receiveState() {
// Receive a state from <host, port> and return
it.
...
}
}
```

Figure 4.c: Implementation of mobility service

The *arrive* method is implemented as follows:

- The *arrive* method calls the *receiveAndRestore* method (figure 4.b).
- The *receiveAndRestore* method is adapted using an instance of the *MyReceiver* class.
- The *MyReceiver* class implements the *ReceiveStateInterface* interface and therefore provides a *receiveState* method, this method aims at establishing a connection to a machine and receiving a *ThreadState* object using de-serialization (figure 4.c). The classes associated with this *ThreadState* object are received relying on the Java dynamic class loading service.

We can also imagine *go* and *arrive* methods that rely on the Wireless Application Protocol instead of IP in order to perform thread migration between JVM installed on wireless hosts [WAPFactory00].

java.lang.threadpack
Class PersistentThreadManagement
public final class **PersistentThreadManagement** extends Object
The PersistentThreadManagement class provides several useful services for making Java threads persistent.

Method Summary	
static void	store(Thread thread, String fileName) Saves the state of the thread argument in the file specified by the name argument.
static Thread	load(String fileName) Restores the execution of a Java thread from the state stored in the file specified by the name argument.

Figure 5: Service for the persistence of Java threads

In the same way, the *PersistentThreadManagement* provides several services for the persistence of Java threads. A part of its application programming interface is illustrated by figure 5. The *store* method saves the current state of a Java thread in a file specified by a name and the *load* method restores a Java thread from a state saved in a file identified by a name. These two methods are also implemented using our *captureAndSend* and *receiveAndRestore* generic methods.

Finally, the *MobileThreadManagement* and *PersistentThreadManagement* classes are two possible adaptations of our generic service of Java thread state capture/restoration. In the same way and for a particular application, our generic service can be adapted to build tools that meet application's needs.

4. Evaluation

This section first presents the performance figures of our thread state capture/restoration service. The cost of migrating a Java thread between two machines and the cost of checkpointing/recovering a thread are then presented. Finally, a comparison between the results of benchmarking our extended JVM and the standard JVM is described. Our evaluation environment is as follows:

- JDK 1.2.2,
- Solaris 2.6, Sun Ultra-I (Sparc Ultra-I 167 MHz),
- Ethernet 100Mb/s.

4.1. Basic costs

The time spent in capturing/restoring a Java thread state depends on the size of the state at capture time. The size of a Java thread depends on the number and the size of the frames pushed onto the Java stack associated with the thread. In the following, we focus our attention on the influence of the number of frames on the cost of our services. In order to vary the number of frames pushed onto thread's Java stack, we used a recursive program (the factorial function).

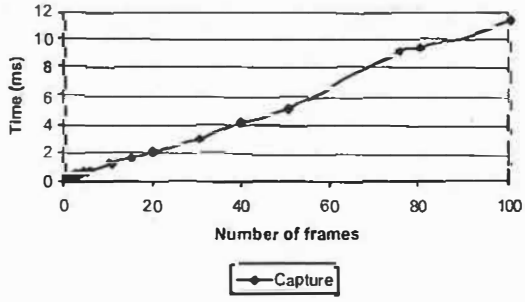


Figure 6: Thread state capture

Figure 6 describes the variation of the cost of a thread state capture operation according to the number of frames on thread's Java stack at capture time. The cost of a capture operation is less than I ms when the number of frames is lower than 10. This cost reaches 2 ms when the number of frames is 20 and 9 ms when the number of frames is 80.

Figure 7 presents the cost of a thread state restoration operation when varying the number of frames on thread's Java stack at capture time. The curve shows that the cost of a restoration operation is less than 1 ms when the number of frames is lower than 80.

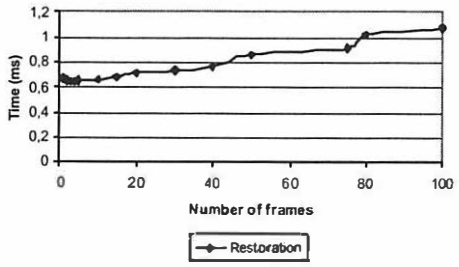


Figure 7: Thread state restoration

Finally, the costs of the capture and the restoration of Java thread's state are acceptable, especially in the case of threads with few frames on the Java stack.

4.2. Evaluation of thread migration

We measured the cost of a Java thread migration based on our thread mobility service. In figure 8, the solid curve represents the variation of the cost of a Java thread migration operation according to the number of frames on thread's Java stack at migration time. The dotted curve represents the cost of a thread state transfer between two machines when varying the number of frames on thread's Java stack.

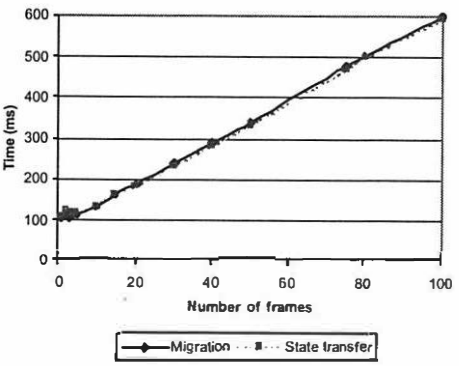


Figure 8: Thread migration

The cost of a thread migration linearly varies from 100 ms to 600 ms when the number of frames on the thread's stack is between 1 and 100. This cost may seem significant but it is mainly due to the cost of thread state transfer, as shown by the two almost superimposed curves. In fact, thread migration consists in capturing thread's state, sending this state to a destination machine and restoring the state in a new thread on the destination machine. So state transfer represents 98% of the total cost of thread migration.

On the other hand, the transfer of a thread state to a destination machine consists in first serializing the state object in order to translate the object graph to a byte array, then transferring the resulting array of bytes over the network to the destination machine and finally de-serializing the byte array on the destination machine in order to rebuild the object graph. The state transfer time can partly be reduced using Java externalization rather than serialization. Externalization allows the application programmer to write its own object transfer policy by only saving information necessary for rebuilding object graphs. Externalization may be until 40% faster than serialization [Sun00c].

4.3. Evaluation of thread checkpointing and recovery

Besides thread migration, we also measured the cost of checkpointing a running Java thread and saving its state on disk and the cost of recovering an execution from a state previously stored on disk. In figure 9, the solid curve represents the variation of the cost of a Java thread checkpointing operation according to the number of frames on the thread's Java stack at checkpointing time. The dotted curve represents the cost of writing a thread state on disk according to the number of frames on the thread's Java stack. In figure 10, the solid curve represents the cost of a Java thread recovery and the dotted curve illustrates the cost of reading a thread state from disk.

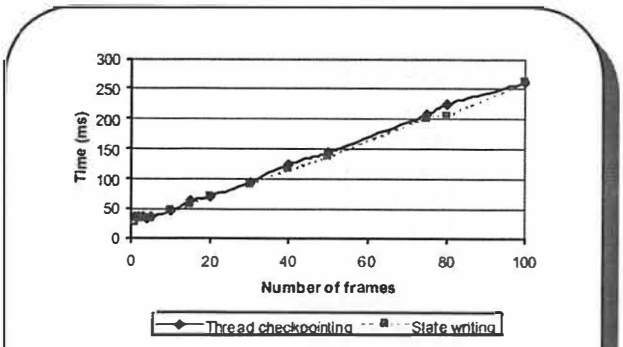


Figure 9: Thread checkpointing

We notice, on the one hand, that the cost of thread checkpointing and the cost of thread recovery increase linearly when the number of frame on the thread's Java stack increases. On the other hand, 97% of the time of thread checkpointing is spent in writing thread's state on disk and 99% of the time of thread recovery is spent in reading thread's state from disk. As explained in section 4.2, the costs of serialization/de-serialization can be decreased by using externalization. The

performance of thread checkpointing can also be improved by performing asynchronous disk writing.

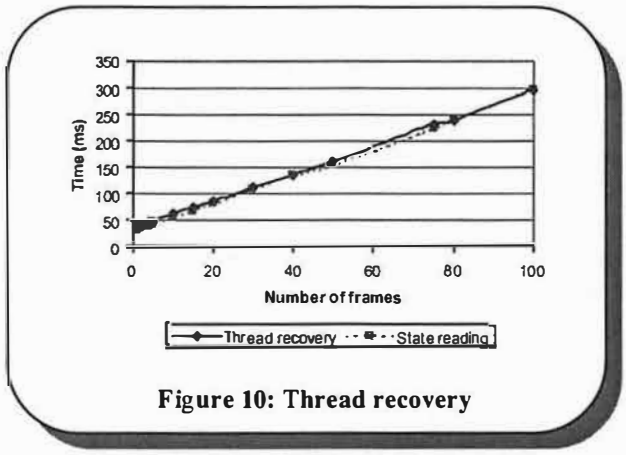


Figure 10: Thread recovery

4.4. Benchmarking the JVM

Our thread mobility and thread persistence services were integrated into the JVM. In order to evaluate the performance of our extension of the JVM, we compared them to the performance figures of the standard JVM. We used two benchmarks: the Embedded CaffeineMark 3.0 general Java benchmark [Pendragon99] and the SciMark 2.0 numeric Java benchmark [Pozo00]. In order to measure JVM performance, the benchmarks were run by disabling JIT compilation.

Tests	Standard JDK 1.2.2	Extended JDK 1.2.2
Overall Score	**1913**	**1913**
Sieve	1447	1457
Loop	3314	3311
Logic	5403	5265
String	1922	1985
Float	1130	1134
Method	871	858

Figure 11: Benchmarking the JVM with Embedded CaffeineMark 3.0

Embedded CaffeineMark consists of 6 tests: finding prime numbers, loops, logic tests, String and Float tests and method calls. The overall score is the geometric mean of the individual scores, i.e., it is the 6^{th} root of the product of all the scores. The score for each test is proportional to the number of times the test was executed divided by the time taken to execute the test, i.e. a higher number represents a better score. Figure 11 presents the results of benchmarking the standard JDK 1.2.2 and our extended JDK 1.2.2. It shows that our extension does not induce any loss of performance on the JVM.

Tests	Standard JDK 1.2.2	Extended JDK 1.2.2
Composite Score	**8.9891**	**9.0977**
FFT (1024)	7.4484	7.6727
SOR (100x100)	18.6662	18.7242
Monte Carlo	0.9157	1.1166
LU (100x100)	7.4484	7.6727
Sparse matmult (N=1000, nz=5000)	7.7224	7.5683

Figure 12: Benchmarking the JVM with SciMark 2.0

SciMark 2.0 is a Java benchmark for scientific and numerical computing. It consists of five computational kernels: Fast Fourier Transform (FFT), Jacobi Successive Over-relaxation (SOR), Monte Carlo integration, dense LU matrix factorization and Sparse matrix-multiply. The kernels are chosen to provide an indication of how well the underlying JVM performs on applications utilizing these types of algorithms. The problems sizes are purposely chosen to be small in order to isolate the effects of memory hierarchy and focus on internal JVM and CPU issues. This benchmark reports a composite score in approximate Mflops (Millions of floating point operations per second). Figure 12 shows the performance figures resulting from benchmarking the standard JDK 1.2.2 and our extended JDK 1.2.2. It shows that our extension does not induce any loss of performance on the JVM.

5. Experimentation

In this section, we describe two experiments that use our mobility service. The first experiment shows the usefulness of strong mobility and the second experiment shows how to build a dynamic reconfiguration tool on top of our mobility service. Finally, we discuss some implementation issues and solutions.

5.1. Strong mobility: Mobile recursive *Fractal*

Two degrees of application mobility can be distinguished: *weak mobility* and *strong mobility* [Fuggetta98]. With weak mobility, only data state information and application's code are transferred. Therefore, on the new location, the mobile application has its actualized data but restarts execution from the beginning. With strong mobility, the code of the application and the state of data and execution are transferred: the application on the destination location

resumes the execution at the point where it was interrupted on the source location.

The usage of weak or strong mobility depends on application's needs. Let's consider a recursive Java application. The recursive calls are translated by a succession of frames on the Java stack associated with the underlying thread. How is this application made mobile?

- Weak mobility does not consider the state of execution (thread's state), so frames previously pushed onto the Java stack are lost after the transfer and the execution restarts from the beginning.
- Strong mobility captures the state of execution and allows the execution to be resumed after the transfer.

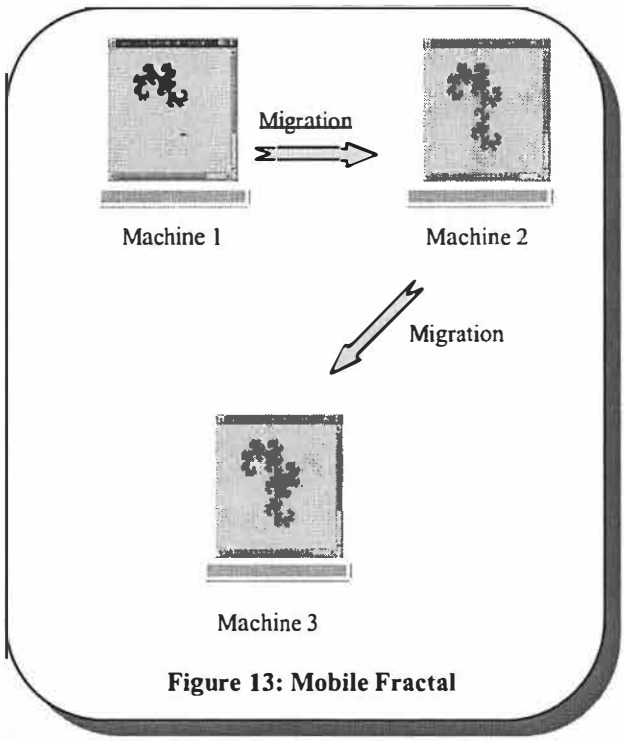


Figure 13: Mobile Fractal

We considered a recursive graphical Java application: the *Dragon* fractal curve where a small dragon appears at a certain depth of recursion [Mandelbrot75]. We implemented a Java *Dragon* application and used our thread mobility service in order to move the application, when it is running, between several machines. Figure 13 illustrates this experiment. The *Dragon* application is first started on a first machine, then moved to a second machine where it resumes its execution and finally moved to a third machine where it finishes its execution. The transfer of the thread calculating the fractal is performed by an external thread that calls the *go* method of our *MobileThreadManagement* class.

5.2. Dynamic reconfiguration: Mobile *Talk*

In this section, we describe how our mobility service can be combined with other Java services (object serialization, dynamic class loading) in order to build a dynamic reconfiguration tool.

We consider a *Talk* application where two remote users exchange messages. Initially, each user starts an instance of the *Talk* application on its personal computer with a graphical user interface. Each user has two communication channels: an input channel to receive messages from the remote user and an output channel to send messages to the remote user. During the talk, one of the users decides to transfer its application to a minimal host with limited physical characteristics (a mobile phone for example) and to resume its execution. This dynamic reconfiguration of the *Talk* application is illustrated by figure 14 and has the following requirements:

- Moving a running application from one host to another.
- Handling communication channels during transfer.
- Replacing the graphical user interface by a textual user interface when arriving on the destination host because of the limited physical characteristics.

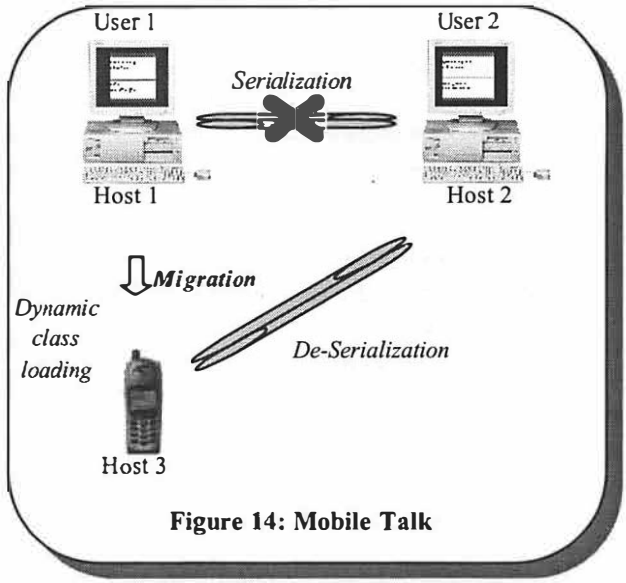


Figure 14: Mobile Talk

To transfer the running mobile *Talk* application to a new host, our mobility service can be used: it takes the current state of the application into account. To transfer the application to a mobile phone, the mobility service must use the Wireless Application Protocol (WAP) [WAPfactory 00].

To tackle the problems of communication channels and user interface, the Java serialization and dynamic class loading services are adapted. In fact, our mobility service relies on both serialization and

dynamic class loading to respectively transfer the objects and the classes used by the application at migration time. These two features can be specialized as follows:

- The serialization of the communication channels can be adapted in order to send a particular message to the remote user informing him about the next migration and then to close the connections. Symmetrically, the de-serialization of the communication channels can be adapted in order to recreate new channels and reestablish the connection with the remote user.
- The dynamic class loading can be adapted in order to use a textual user interface rather than the graphical one on the mobile phone.

Finally, the combination of our mobility service, the serialization and the dynamic class loading enables the building of a complete dynamic reconfiguration tool. This application has been experimented with a prototype implementation on our extended JDK 1.2.2. A port of our services to the K Virtual Machine, a lightweight JVM, is planned [Sun00b].

5.3. Discussion

In this section, we discuss some issues encountered when implementing thread mobility and thread persistence. Let's focus our attention on the mobility of a thread:

- What happens if a thread moves from a source host to a destination host while it is using objects shared with other threads on the source host?
- How are the communication channels connecting several threads handled when one of these threads moves to a new host?
- What happens if a thread that belongs to a multi-threaded application move to a new host?

We now tackle each of these issues and propose possible solutions.

What happens if a thread moves from a source host to a destination host while it is using objects shared with other threads on the source host? A first solution consists in replicating the shared object and transferring it with the mobile thread [Garcia-Molina86]. In this case, the consistency of the replicas must be managed. Another solution to the problem of shared objects is to use proxies on the destination host in order to allow remote access to shared objects. A problem of object availability occurs if the source host crashes [Chou83].

How are the communication channels connecting several threads handled when one of these threads moves to a new host? A first approach consists in using proxies on the destination host in order to access the communication channels remotely. A problem of channel availability occurs if the source host crashes. Another approach consists in closing the channels on the source host and recreating them on the destination host. In this case, messages in transit must be redirected to the new location and the naming of the new channels must be actualized on other hosts.

What happens if a thread that belongs to a multi-threaded application move to a new host? The thread can move alone to the destination host and communicate with the other threads remotely, or it can move with all the other threads or with a sub-set of them.

Finally, for each of the discussed issues, the solution strongly depends on application's needs. That is why we deliberately chose not to impose a particular solution at the level of our thread mobility and thread persistence services. The programmer of the application is thus free to choose the more appropriate approach.

6. Related work

Many systems have been developed providing process mobility and persistence, considering either homogeneous or heterogeneous processor architectures. There are a number of surveys discussing these features [Milojicic97] [Deconinck93]. Both mobility and persistence of control flows are based on a mechanism that enables the capture and the restoration of executions' state. Let's focus our attention on such mechanisms in the Java environment.

Three main approaches to address the problem of capturing/restoring the state of Java threads are distinguished: an *explicit* approach, an *implicit* approach based on a pre-processor of the application code and an *implicit* approach based on an extension of the JVM.

In the first approach, which we call *explicit management*, the programmer of an application has to entirely manage the capture and the restoration of the state of his application. For this purpose, the programmer has to explicitly add supplementary code in fixed points of his program and usually has to manage his own program counter. The added code manages a backup object in which information relative to the state of the application is stored. The backup object is then used in order to restore the application execution. When restoring the state of the application, the first statement of the program is a branch to the point where the program must continue. This approach is not flexible and implies a modification of the application itself if new backup points are added. This approach is used in most of applications based on mobile agent platforms [Chess95] that provide weak

mobility, such as Aglets [IBM96] and Mole [Baumann98].

The two other approaches, which we call *implicit*, provide a transparent service for capturing/restoring thread state. The service is independent from the application code and is provided as a function that may be called by the application itself or by an external application. These two approaches differ by their implementation:

- The first implicit approach consists in pre-processing the source (or byte) code of the application in order to insert statements. The inserted code attaches a backup object to the application. While the application is running, the backup object is re-actualized with the state of the application. When an application requires a snapshot of its state, it just has to use the associated backup object. In order to restore the execution state, data stored in the backup object are used to re-initialize the application in the same state as at snapshot time. This restoration is achieved by re-executing a different version of the application code (produced by the pre-processor) in order to rebuild the stack and re-initialize the local variables with the values stored in the backup object. The main motivation of this approach is that it does not modify the JVM. But its drawback is that it induces a significant overhead on application performance due to the inserted code, and on execution restoration which requires a partial re-execution of the application. The Wasp project provides a Java mobile agent platform based on a pre-processor which instruments the source code of Java applications [Fünfrocken98]. Several implementations of Java thread mobility based on a pre-processor of the bytecode are proposed [Truyen00] [Sakamoto00].
- The second implicit approach consists in extending the JVM in order to make threads' state accessible from Java programs. This extension must provide a facility for extracting the thread state and storing it in a Java object. The extension must also provide a facility for building a new thread initialized with a previously captured state. These facilities can only be used on extended virtual machines. We followed this last approach for two reasons:
  - It reduces the overhead on application performance (no inserted code) and reduces also the cost of the capture/restoration service (its implementation is mainly native).
  - Since the thread state capture/restoration service has many applications, we believe that it is a basic functionality which must be integrated within the JVM.

  This solution has been used in the implementation of the Sumatra mobile agent platform [Ranganathan97]. Unlike Sumatra which supplies a mobility service, our implementation provides a generic service intended for other uses than mobility, like persistence [Bouchena99]. The recently proposed Merpati system also follows this approach [Suezawa00]. It makes the whole JVM mobile or persistent, with all its threads, while our services are fine-grained and can be applied to one thread.

To summarize, our services provide a transparent and fine-grained Java thread state capture/restoration facility. They can be used for several purposes among which thread mobility and thread persistence. They are integrated into the JVM and thus present competitive performance figures. A comparison between the performance of the first implicit approach (Wasp's mobility service) and the second implicit approach (our mobility service) can be found in [Bouchenak00].

7. Conclusion and future work

Since the Java virtual machine does not provide any access to the state of Java threads, we designed and implemented a new service for the capture and restoration of thread state. Our capture/restoration service is generic: we used it as a basis for the implementation of thread mobility and thread persistence services. With these services, a running Java thread can, at an arbitrary state of its execution, migrate to a remote machine or be checkpointed on disk and then recovered. In addition, the migration or the checkpointing of a thread can be initiated by the thread itself or by another thread.

Our services were integrated into the JVM, so they provide acceptable performance figures without inducing overhead on JVM performance. Finally, we experimented with a prototype implementation a dynamic reconfiguration tool based on our mobility service and applied to a running distributed application.

The lessons learned from this experiment are that:

- It is possible to extend the Java virtual machine with thread mobility and persistence services without re-designing the whole JVM.
- This implementation provides reasonable and competitive performance costs.

At the present time, we are considering the usage of our services in real world applications such as dynamic load balancing in distributed systems and the integration of our services into distributed Java virtual machines. We also plan to port our services to the K Virtual machine, the lightweight JVM, in order to make

them available on small devices such as phones and PDA [Sun00b].

Acknowledgments

I would like to thank Sacha Krakowiak and Jacques Mossière for providing many useful suggestions that significantly improved this paper.

References

[Ambler99] S. W. Ambler. The Design of a Robust Persistence Layer For Relational Databases. *AmbySoft Inc. White Paper*, October 1999. http://www.ambysoft.com/persistenceLayer.html

[Baumann98] J. Baumann, F. Hohl, M. Straber and K. Rothermel. Mole - Concepts of Mobile Agent System. *WWW Journal, Special issue on Applications and Techniques of Web Agents*, volume 1, no 3, 1998. http://mole.informatik.uni-stuttgart.de/

[Bouchenak00] S. Bouchenak and D. Hagimont. Pickling threads state in the Java system. *Proceedings of 33rd International Conference on Technology of Object-Oriented Languages (TOOLS Europe'2000)*, Mont-Saint-Michel, France, June 2000. http://sirac.inrialpes.fr/~bouchena

[Chess95] D. Chess, C. Harrison and A. Kershenbaum. Mobile Agents: Are They a Good Idea? *T.J. Watson Research Center White Paper*, IBM Research Division, March 1995. http://www.research.ibm.com/iagents/ publications.html

[Chou83] T. C. K. Chou and J. A. Abraham. Load Redistribution under Failure in Distributed Systems. *IEEE Transactions on Computers*, volume 32, no 9, September 1983.

[Deconinck93] Geert Deconinck, Johan Vounckx, Rudi Cuyvers and Rudy Lauwereins. Survey of Checkpointing and Rollback Techniques. *Technical Report, Katholieke Universiteit Leuven*, Belgium, June 1993.

[Douglis92] F. Douglis and B. Marsh. The Workstation as a Waystation: Integrating Mobility into Computing Environment. *The Third Workshop on Workstation Operating System (IEEE)*, Key Biscayne, Florida, USA, April 1992. http://www.douglis.org/fred

[Fuggetta98] A. Fuggetta, G. P. Picco and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, volume 24, no 5, 1998. http://www.cs.ucsb.edu/~vigna/listpub.html

[Fünfrocken98] S. Fünfrocken. Transparent Migration of Java-based Mobile Agents(Capturing and Reestablishing the State of Java Programs). *Proceedings of Second International Workshop Mobile Agents 98 (MA'98)*, Stuttgart, Germany, September 1998. http://www.informatik.tu-darmstadt.de/~fuenf

[Garcia-Molina86] H. Garcia-Molina. The Future of Data Replication. *Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles, California, USA, January 1986.

[Gosling96] J. Gosling and H. McGilton. The Java Language Environment. *Sun Microsystems White Paper*, May 1996. http://java.sun.com/docs/white

[Hofmeister93] C. Hofmeister and J. M. Purtilo. Dynamic Reconfiguration in Distributed Systems : Adapting Software Modules for replacement. *Proceedings of the 13th International Conference on Distributed Computing Systems*, Pittsburgh, USA, May 1993.

[IBM96] IBM Tokyo Research Labs. *Aglets Software Development Kit*, 1996. http://www.trl.ibm.co.jp/aglets

[Lindholm96] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison Wesley, 1996.

[Mandelbrot75] B. Mandelbrot . *Les Objets fractals : forme, hasard et dimension*. Flammarion, 1975.

[Milojicic97] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler and S. Zhou. Process Migration. *TOG Research Institute Technical Report*. March 1999. http://www.camb.opengroup.org/RI/java/moa

[Milojicic99] D. Milojicic, F. Douglis and R. Wheeler. *Mobility: Processes, Computers and Agents*. Addison Wesley, February 1999.

[Nichols87] D. A. Nichols. Using Idle Workstations in a Shared Computing Environment. *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, Austin, Texas, USA, November 1987.

[Oueichek96] I. Oueichek. *Conception et réalisation d'un noyau d'administration pour un systéme réparti à objets persistants*. Thèse de Doctorat, Institut National Polytechnique de Grenoble, France, October 1996.

[Pendragon99] Pendragon Software Corporation. CaffeineMark™ 3.0. *CaffeineMark 3.0 documentation*, 1999. http://www.pendragon-software.com/pendragon/cm3

[Pozo00] R. Pozo and B. Miller. SciMark 2.0. *SciMark 2.0 documentation*, 2000. http://math.nist.gov/scimark2

[Ranganathan97] M. Ranganathan, A. Acharya, S. D. Sharma and J. Saltz. Network-aware Mobile Programs. *Proceedings of the USENIX Annual Technical Conference*, Anaheim, California, USA, January 1997. http://searchpdf.adobe.com/proxies/0/38/54/9.html

[Sakamoto00] T. Sakamoto, T. Sekiguchi, A. Yonezawa. *Bytecode* Transformation for Portable Thread Migration in Java. *Proceedings of Second International Workshop Mobile Agents 2000 (MA'2000)*, Zurich, Switzerland, September 2000. http://web.yl.is.s.u-tokyo.ac.jp/~takas/

[Suezawa00] T. Suezawa. Persistent Execution State of a Java Virtual Machine. *Proceedings of the ACM 2000 Java Grande Conference*, San Francisco, California, USA, June 2000. http://www.ifi.unizh.ch/staff/suezawa/

[Sun00a] Sun Microsystems. Java 2 SDK, Standard Edition. *Sun Microsystems documentation*, 2000. http://java.sun.com/products/jdk/1.2

[Sun00b] Sun Microsystems. Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices. *Sun Microsystems White Paper*, 2000. http://java.sun.com/products/cldc

[Sun00c] Sun Microsystems. Improving Serialization Performance with Externalizable. *Technical Tips*, Sun Microsystems, April 2000. http://developer.java.sun.com/developer/TechTips/2000/tt0425.html

[Truyen00] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen and P. Verbaeten. Portable Support for Transparent Thread Migration in Java. *Proceedings of Second International Workshop Mobile Agents 2000 (MA'2000)*, Zurich, Switzerland, September 2000. http://www.cs.kuleuven.ac.be/~eddy/research.html

[WAPfactory00] WAPfactory. *Wap.com*. 2000. http://www.wap.com/

[Wojcik95] Z. M. Wojcik and B. E. Wojcik. Optimal Algorithm for Real-Time Fault Tolerant Distributed Processing Using Checkpoints. *Informatica*, volume 19, no 1, February 1995. http://ai.ijs.si/informatica/

Bean Markup Language: A Composition Language for JavaBeans Components

Sanjiva Weerawarana, Francisco Curbera, Matthew J. Duftler,
David A. Epstein*, and Joseph Kesselman
Component Systems Group
IBM TJ Watson Research Center
Hawthorne, NY 10598
{sanjiva,curbera,duftler}@us.ibm.com, depstein@vastvideo.com, keshlam@us.ibm.com

Abstract

Although the benefits of software component composition are today widely accepted, component oriented software development is not yet as widespread as its multiple advantages may suggest. This is so in spite of the maturity reached by several component models (Microsoft's COM, JavaBeans, OMG's CORBA), and their general acceptance by large communities of developers. Thus, while components are being 'used' in software development, the development process itself is not fully component oriented. One major roadblock limiting the adoption of a component oriented development process is the lack of viable component composition languages. This paper introduces a component composition language specifically designed for the composition of JavaBeans components.

The Bean Markup Language (BML) supports component composition in a first-class manner. BML has language constructs for describing inter-component bindings, for constructing aggregates of components, for macro expansion and for implementing certain types of recursive compositions. Further, it allows the specification of "glue code" in any traditional scripting language (for example, JavaScript) to enable powerful adaptation during composition.

*Presently Vice President, Development & Architecture, VastVideo Inc., Astoria, NY 11106.

1 Introduction

The benefits of software component composition are today widely accepted, see [6, 7, 21, 11, 2, 15], however, component oriented software development is not yet as widespread as its multiple advantages may suggest. This is so in spite of the maturity reached by several component models (Microsoft's COM, JavaBeans, OMG's CORBA), and their general acceptance by large communities of developers. Thus, while components are being "used" in software development, the development process itself is not fully component oriented. One major roadblock limiting the adoption of a component oriented development process is the lack of viable component composition languages. As has been argued in [13] and [21], component-oriented development is likely to be much more successful when first-class mechanisms enabling simple forms of composition are used.

Component-oriented development is a natural evolution of the object-oriented development paradigm. Components provide a programming abstraction in terms of events, properties and methods. The properties and methods of a component allow the component to be configured and events are how the component communicates interesting information to its consumers. In the component-oriented development model, application development becomes a matter of "scripting together" a set of such components, where the components themselves are sometimes bought from a collection of third parties and sometimes developed in-house. This type of composition enables loose coupling and provides the necessary hooks for adapting pre-built components as needed to form the desired aggregation. Object-oriented development is clearly the predominant methodology used in developing the

components themselves [21, 11].

The key technologies that enable component-oriented development are a component model and a composition mechanism. A component model is a set of conventions and a run-time architecture that provide an environment to define and manipulate software components. The definition of a software component varies. However, a common theme is that it is an executable, self-contained, dynamically loaded/bound module that exhibits certain types or interfaces or contracts to other components that adhere to the same component model [7]. The three most popular component models in use today are: Microsoft's COM [5], OMG's CORBA [18] and JavaSoft's JavaBeans [20]. The work described in this paper assumes the JavaBeans component model, but it could be implemented with any of these models.

Component composition is the key programming task required and enabled by component-oriented development [2]. Component composition is the process of creating component instances, configuring them and putting them together to form composite components or applications. Configuring components consists of manipulating their properties and also invoking their methods. Putting components together consists of describing component–to–component relationships as well as aggregations such as component hierarchies. An ideal language for component composition would have first-class syntax and semantics to support such composition operations.

Component composition can be performed in a variety of ways. The obvious way is to use a traditional programming language to write code that creates instances of components and composes them together by using the appropriate method calls. This task is commonly done by the "main" procedure of an application or that of a composite component.

Traditional programming languages are however not the best suited for component composition. Since their syntaxes and semantics do not support component composition concepts in a first-class manner, composition operations are supported using other existing language elements like, say, method calls. As a result, the composition operations are lost amongst the rest of the code and the compositional structure is obscured. A discussion of the shortcomings of object oriented languages when applied to component composition can be found in [1].

A second common approach to software composition is the use of scripting languages. Scripting languages are programming languages which are supposed to be in some sense "easier" to program with: they are typically loosely typed and interpreted. They are commonly used for application prototyping, configuration, customization and extension [17, 11]. Scripting is a natural counterpart to component-oriented development - components can be written in standard object-oriented languages and then "glued together" to form applications. However, as a mechanism for component composition, scripting languages such as PERL [22], Tcl [16] and JavaScript [8] suffer from the same problem as traditional programming languages, their lack of first class support for composition. In fact, scripting languages do not add abstractions to programming; their primary goals are to reduce complexity by eliminating syntax and types to make programming "easier," not to change the level of abstraction.

Visual composition is another popular approach to component composition. A visual builder allows one to select components from a palette, place them on a composition editor and visually "wire" together the component interactions. Where required, additional behavior can be added using scripts, for instance, to intercept an event on its path from a source to a target and trigger special actions. The JavaBeans component model in fact recognizes the role of visual builders and provides for the component to distinguish between build-time and run-time. Using this, a JavaBeans component may, for example, present a build-time user interface that can be used to configure the component. Visual composition clearly provides first-class support for the composition operations described earlier. Visual composition's power is also its failing however: it is interactive and graphical, and, consequently, not an option in scenarios where the composition is done by a non-interactive mechanism. This is the case, for instance, when user interfaces are automatically generated from data input specifications. A first class representation of the composition would allow both generation methods (interactive and non-interactive) to interoperate, by acting as a neutral intermediate format. Moreover, if appropriately designed, the intermediate format could also be directly manipulated by developers.

Hence, a solution to the shortcomings of existing compositions techniques is to introduce a component composition language, that is, a language in

which the basic component composition operations is supported in a first class manner.

Extending the syntax and semantics of existing languages looks like an attractive option, but it can be argued that a special purpose composition language will do a better job capturing the specific nature of composition operations. Moreover, it is noted in [1] that object-oriented languages like Java and object oriented design tend to be used to produce domain specific designs, rather than standard architectures more suitable for the kind of reuse expected from software components.

The most relevant effort in the definition of a composition language is probably Piccola [1], a declarative composition language founded on a variant of the π calculus, in which components are viewed as interacting processes. Piccola is a very small language, which is able to support a variety of component models through the definition of different component composition ("architectural") styles. Piccola is an on-going research effort.

The introduction of a new language has some major practical problems, though. It requires retraining and the development or adaptation of tools to support it. Furthermore, component models and run-time models to interact with other languages must be developed. Thus, an approach where a new language, a new component model and a new run-time is needed is not immediately suitable as a mechanism to enable component-oriented development in practice.

Hence, the problem we are interested in can be formulated as follows:

How to design a component composition language which can be seamlessly integrated in today's software development environment.

We believe that an answer to this problem can be a key to successfully driving today's development methodology toward the component oriented development paradigm. This paper describes an answer to this problem, the Bean Markup Language (BML), a declarative language for the composition of JavaBeans components.

The rest of the paper is organized as follows. Section 2 states the requirements for a composition language that can capture the design problem stated above. Section 3 describes the design of the BML language and how BML addresses these requirements. Section 4 describes the BML implementation and run-time support. Finally, in Section 5 we address open problems and research issues.

2 Requirements for a Composition Language

The purpose of this section is to map the design problem stated in the Introduction to a list of design requirements. The starting problem has two parts: designing a composition language, and assuring its easy integration in development environments.

Several papers have dealt with the problem of specifying requirements for successful composition languages. The following discussion owes much to the ones found in [13], [14] and [3].

Our first requirement states a list of composition operations that a composition language should support.

1. The following **composition operations** must be supported by the language:
   - **Binding communication channels.** Communication channels let components exchange data and invoke behavior. Good examples are pipes and filters, and event notification in JavaBeans.
   - **Creating higher level component aggregates.** In this operation components are combined to produce higher order functional constructs. The combination typically involves creating a hierarchy of components, as when creating graphical user interfaces.
   - **Macro expansion** of parametrized components. Macro expansion can be used in several ways to compose components. In the COMPOST language, [3], source components are connected together by expanding (binding) "generalized program elements" present in each component's code. Another case of composition by macro expansion is described in [4] and [19] discusses a mechanism to maintain correct scoping while generating programs.

- **Recursive component composition.** Component composition is used to create new components, rather than an application. This is a powerful technique that enables components to become software abstractions at different levels, and provides support for top-down progressive refinement design strategies. It is also has an important role providing scalability to the language, since it allows using the same language composition abstractions at different configuration levels.

A language solely devoted to component composition must also provide effective separation of concerns between the person doing the composition and the developer of components. This is nothing but a restatement of the principle that the composition of components must require no knowledge of their implementations. In particular, the language must provide a way to address "compositional mismatches", i.e. situations when the interfaces of two components are incompatible and don't allow direct composition.

2. The language should allow the specification of "glue code" to deal with compositional mismatches.

Glue code provides the bridge through which the two interfaces can interact. In object oriented design this correspond to the "adapter" pattern, [9].

The next requirement deals with the important issue of reusing component application designs.

3. The language should support component frameworks.

Here the notion of a component framework is similar to the frameworks found in object oriented design, see [9, 10] for instance. It is defined in [21] as a software architecture that provides basic relationships among components and allows instances of those components to be plugged in the framework. Frameworks are important tools that provide component assemblers with the infrastructure needed to build structured applications. Frameworks are also important as a knowledge sharing mechanism and as enablers of large scale component oriented development.

In order to assure seamless integration of the language into current development environments, we state in our requirements list the need for low adoption costs, and the ability to reach different development platforms as possible:

4. Reduce to a minimum the learning process for the language. In particular, use whenever possible existing languages, syntactic and semantic conventions. The Java language and the XML syntax would be good starting points according to this criterion.

5. Eliminate the need for new support tools, whenever possible. Existing development environments should be able to provide support for the language with minimal investment.

6. The language primitives must allow easy extension to support alternative component models. While the focus of this work has been the JavaBeans model, it should enable a direct extension to support the COM and CORBA models.

3 BML: A Composition Language for JavaBeans

The BML language was designed to meet many of the requirements we have identified in the previous section. This section describes the BML language, its design principles, and some of its most relevant features.

This section is organized as follows: first we explain some of the general design decisions behind BML. Finally, we describe the major language elements and explain the support that BML provides for the composition of components and other relevant features of the language.

Design Principles

BML has been designed as an XML–based declarative language for describing the composition of JavaBeans applications. This statement summarizes three major design principles, which we review in this section.

XML syntax. BML intentionally de-emphasizes the importance of syntax. From the two alterna-

tives of choosing a syntax with multiple elements and structures (e.g., a Java–like syntax), or following a relatively "syntax–free" approach (e.g., the Lisp way), the second option was judged more likely to allow the language to satisfy requirements 4 and 5 from Section 2. This is the reason why XML was chosen as the syntactic model for the language. Its XML syntax is in fact the main reason why BML complies with those two requirements.

XML languages follow a relatively simple syntax model (see [12]), and are described using the XML DTD [24] or the XML Schema [25] metalanguages. The XML model allows very limited syntactic options, essentially the choice of whether to use an XML attribute or an XML element to represent features of the language. XML, on the other hand, is already a widely embraced industry standard, its simple syntax is well known by many developers, and supporting middleware is available for all major computing platforms.

While a Java–like syntax would have the advantage of providing a certain degree of familiarity to Java developers, it would also have the disadvantage to being only Java–*like*, and not exactly Java. In fact, the intended user of BML is the component composer, who may not even be a Java developer.

A declarative language. BML is designed to describe the composition of a set of components rather than to describe how the composition is to be implemented. To fully understand this distinction we state here the four phases of the component development process:

1. **Authoring–time**. Components are created, typically using an object oriented language, and packaged for use by third parties.
2. **Composition–time**. This is design–time, when components are selected, configured and the desired composition is described. The role of component languages is to capture this composition.
3. **Assembly–time**. Part of the application startup time. The composition described in the component language script is realized into an executable application, typically by a component language processor.
4. **Run–time** After the composition is performed, the application runs to perform its function. Non-compositional processing happens at this time, typically by executing the component's own code.

The role of BML is to represent the structure of a composed application as designed at composition time. The actual assembly of the components is the role of the language processor at assembly time. BML defines an assembly–time environment to support this distinction (the assembly–time environment is described later in this section and in Section 4). This is the reason why we describe BML a as declarative composition language. It must be remarked that, while BML allows the inclusion of sections of "glue code" for the purpose of solving compositional mismatches and "configuration instructions" for configuring individual components for composition, the BML language's compositional elements are declarative.

Application versus component composition. Compositions can be either "final" or reusable. Final compositions are *applications*. Reusable compositions are themselves components and can be used in new compositions, both final and reusable. The main difference between the two is that reusable compositions present a well defined public interface that identifies them as components in the component model under consideration, and allows reuse. BML is designed to enable the creation both types of compositions, by providing component definition language elements in addition to the basic compositional and configurational elements. The ability to define reusable components in the language provides support for the for recursive composition requirement listed in Section 2.

3.1 BML Language

As an XML based language, BML uses different XML elements for each composition operation. This section presents the BML solution to the composition language problem by describing how it addresses each of the requirements listed earlier. We describe only the essential features of each element in this document; complete documentation can be found in the BML User's Guide, which is part of the BML distribution [23]. The syntax of BML is summarized in a BNF–like form in Table 1.

The role of BML is to capture a composition. A composition script is represented in BML by a <script> element. The contents of this element are

<script>	::=	(S | <cast>)+
<bean>	::=	<args>? S*
<args>	::=	V+
<property>	::=	V?
<field>	::=	V?
<event-binding>	::=	<bean> | <script>
<call-method>	::=	V*
<cast>	::=	V?
<string>	::=	text
<add>	::=	V+
C	::=	(<bean> | <string> | <property> | <field> | <call-method> | <script>)
S	::=	(C | <event-binding> |
	::=	<add>)
V	::=	(C | <cast>)

Table 1: BML Syntax Summary

arbitrary BML elements and the result of evaluating it is the value of evaluating the last child element.

We start the description of the BML language with a small, yet complete example.

The Juggler

This section provides an simple example of a BML application. The purpose is to give the reader an early view of a significant subset of BML.

The example shows how BML can be used to compose a collection of AWT components into an application. The application includes an animation component and two buttons that control it, as well as a window frame component that acts as a container component. Figure 1 shows the resulting application. The example code is shown in Figure 2.

We now briefly explain how the code in in Figure 2 works.

Note that line 0 is the XML declaration which is required of XML documents. In line 2 a new a new script of BML statements is started with the <script> element. The <bean> element in line 3 creates a component of type java.awt.Frame and uses the *id* attribute to assign to it the name "frame". In line 4 the title property of the frame bean is set. The <event-binding> element in line

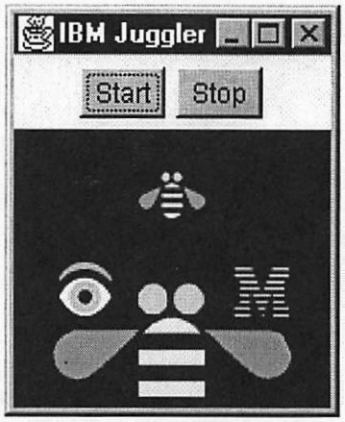


Figure 1: The Juggler Application

5 binds the script in lines 6–10 so it is run when a "window" event occurs and the event is delivered via the windowClosing method. The script contains one statement (lines 7–9) which causes the program to exit.

On line 14 the animator component is created and given the name "Juggler". Lines 13–16 aggregate this component into the container "frame" at the center position using the <add> element. On line 18 a button component is created, its label property is set to "Start" on line 19. Line 20 binds the script on lines 21–23 to be run when an "action" event occurs on the button. The script invokes the start method of the "Juggler" component using the <call-method> element. Observe that the target component is identified using the "target" attribute. Lines 17–27 aggregate the button component into the container "frame" in the north position. Similarly, lines 28-38 create another button component (this one for stopping the animation) and aggregates it to the "frame" component. Lines 40-41 invoke the "pack" and "show" methods of the frame in order to bring it to the screen. Finally, line 43 invokes the "start" method of the animator component to initiate the animation.

In the following sections we review in detail some important aspects of the language.

Naming and Scoping in BML

A mechanism to identify components is fundamental to any composition language. In the previous section we have seen how beans can be created with the

```
0     <?xml version="1.0"?>
1
2     <script>
3       <bean class="java.awt.Frame" id="frame">
4         <property name="title" value="IBM Juggler"/>
5         <event-binding name="window" filter="windowClosing">
6           <script>
7             <call-method target="class:java.lang.System" name="exit">
8               <cast class="int" value="0"/>
9             </call-method>
10          </script>
11        </event-binding>
12
13        <add>
14          <bean class="demos.juggler.Juggler" id="Juggler"/>
15          <string value="Center"/>
16        </add>
17        <add>
18          <bean class="java.awt.Button">
19            <property name="label" value="Start"/>
20            <event-binding name="action">
21              <script>
22                <call-method target="Juggler" name="start"/>
23              </script>
24            </event-binding>
25          </bean>
26          <string value="North"/>
27        </add>
28        <add>
29          <bean class="java.awt.Button">
30            <property name="label" value="Stop"/>
31            <event-binding name="action">
32              <script>
33                <call-method target="Juggler" name="stop"/>
34              </script>
35            </event-binding>
36          </bean>
37          <string value="South"/>
38        </add>
39
40        <call-method name="pack"/>
41        <call-method name="show"/>
42      </bean>
43      <call-method target="Juggler" name="start"/>
44    </script>
```

Figure 2: The Juggler Script

<bean> element, named using the *id* attribute, and located with the *target* attribute. Assigning names to new components is optional, but if a name is assigned then the component is registered in current scope with that name.

BML is a lexically scoped language. A scope is defined by the <script> and the <bean> elements. The default scope is created by the opening <script> element and nested scopes are explicitly created by nesting <script> elements; nested <bean> elements implicitly create a new scope. The scoping semantics are as usual with inside–out visibility and standard shadowing rules. The scope is represented at assembly–time as an "object registry", a registry which provides a name–to–reference mapping and is part of the BML language processor's environment during assembly–time.

A related issue is how BML uses XML's containment model to effect a Pascal–style "with" operator. Recall from our previous example that the *target* attribute is used to name the bean on which an operation is to be performed. In BML the default target for a composition operation is the closest enclosing component, as in the next example.

```
<bean class='java.awt.Button'>
  <property name='label'
            value='Click Me'/>
</bean>
```

Configuration of Components and Type Conversion

JavaBeans components may have configurable properties. BML uses the <property> element for this purpose. The example in Figure 1 shows that the value of the property can be encoded using the *value* attribute, as long as this value can be represented as a string.

When there is no possible string representation, the <property> element is given one child element, the result of evaluating which becomes the value assigned to the property. The following example shows how one can change the layoutManager property of a Panel component:

```
<bean class='java.awt.Panel'>
  <property name='layoutManager'>
    <bean
      class='java.awt.BorderLayout'/>
  </property>
</bean>
```

The <property> element also supports retrieving property values. This is effected by omitting any value for the property; thus, if a value to assign the property is not found, it is treated as an "rvalue" instead of as an "lvalue."

We now discuss type conversions issues arising when property values are encoded as strings. We must note first that this issue is a consequence of the lack of typing information in XML, which results in all the data being encoded as strings.

If the type of the property being set is not a string, a type conversion must be performed before the value can be set. For instance, one may wish to set color–valued properties by giving a string containing the RGB representation of the color. BML's approach is to separate the type conversion problems from the compositional problems as much as possible. All the type conversion logic is considered part of the assembly–time environment of the language, and is not reflected in the BML script.

The BML processor's assembly–time environment contains a registry, called the type converter registry, which is a collection of code that is able to convert data from one type to another. If a type conversion is deemed necessary, the processor will transparently invoke the appropriate converter in order to effect the setting of the property. The type converter registry mechanism serves to improve the declarative nature of BML: it enables one to concentrate on the required composition operations and defer the issues of *how* to realize them until later (and probably to someone else).

Type conversions can also be explicitly requested in BML. The <cast> element is a utility element used to explicitly request a type cast, or to explicitly invoke a type converter to change the type of a value. An example of is provided in line 8 of Figure 1, where a string to integer conversion is requested. Explicit casts are commonly found in BML in the the arguments of a <call-method> invocation (lines 7 to 9 in Figure 1), a common way of performing more complex configuration of components.

Binding Events

Binding communication channels is one of the main composition operations described in Section 2. In the JavaBeans model inter–component composition channels are event streams. In order to do the binding, two requirements must be met:

- The event source must be notified of the listeners' interest in receiving the events.
- Event listeners must be of a suitable type which is statically defined by the event source.

BML uses the <event-binding> element for this purpose, as in the next example:

```
<bean class='java.awt.Button'>
  <event-binding name='action'>
    <bean class='MyActionListener'
          id='a1'/>
  </event-binding>
</bean>
```

Notice that the component 'a1' must be of the appropriate type for this the binding operation to be valid. This kind of binding of communication channels is hence fairly restrictive as the components must be statically designed to be aware of each other's event types. The following section discusses how this is generalized to make event bindings more adaptable.

Writing "Glue" Code

Composing components that are not pre–designed to be linked together often requires the writing of "glue" code to solve these compositional mismatches (recall requirement 2 from Section 2). In the JavaBeans case the problem is even worse because the event binding architecture which requires that the event listener implement a certain interface type.

BML addresses this requirement by allowing the component composer to author glue code in any of several traditional scripting languages. The currently supported languages include JavaScript, Jacl, JPython and VBScript.

This is an important design point. Traditional languages are better fit for writing glue code because, typically, the glue code does not perform component composition, but rather some type of data adaptation to allow components to interact. A composition language is clearly less suited for such tasks than a traditional scripting language, except perhaps for the most elementary ones. Observe also that this further reinforces the clearer separation between component authoring and component composition: while JavaBeans authors are Java programmers, component composers need not be so.

The glue code is directly embedded in the composition script using a <script> element as the child of an <event-binding>. In lines 20 to 24 in Figure 1, for instance, a BML script is provided to cause the invocation of the "start" method when an "action" event is received.

The code in these scripts is executed at run–time when events are generated by the event source component. However, BML provides static scoping for the script, that is, any component that is referenced by the script and was previously registered within its lexical scope will be available during script evaluation. Section 4 describes the implementation of <script>.

Aggregation

Aggregation of components into hierarchies is another major composition operation. BML supports it through the <add> element. The following example illustrates the process of adding a java.awt.Button component to a java.awt.Panel component:

```
<bean class='java.awt.Panel'>
  <add>
    <bean class='java.awt.Button'/>
  </add>
</bean>
```

The meaning of an aggregation operation is defined by the "container" into which aggregation is occurring. This is the default target bean (in XML terms, the parent <bean> element of the <add>), unless otherwise stated by the <add> element. BML's approach is to stay away from differences in the semantics of the operation; only its compositional significance is of interest. Observe that the operation of nesting a <bean> element inside another has very

different semantics from the aggregation operation, since the first one corresponds only to the declaration of a bean inside the parent's scope.

Aggregations defined by different containers may require different data to be specified before the operation can be performed. In the above example, the <add> element has only one child because the layout manager that the panel uses (java.awt.FlowLayout) does not require any other information. However, the add operations included the example from Figure 1 take two arguments, since the default layout manager for the java.awt.Frame component, the BorderLayout, requires that we indicate the layout area in which the a component is to be added. In general, the first child element of <add> is expected to identify *what* to add and any other children are expected to be additional information as needed by the container's semantics for aggregation

The mechanics of how the aggregation is implemented are part of BML's assembly–time environment. This includes a registry (the adder registry) of code fragments (adders) that implement specific aggregation operations for specific container types. The separation of the compositional meaning of the operation from the mechanics of its implementation mechanism serves to further increase the declarative nature of BML: the component composer is only concerned with the desired aggregation structure and not with how that is to be actually realized.

Recursive Composition

Recursive composition of JavaBeans requires a way to define a new component in terms of compositions of beans. The language elements presented so far deal with the connection of already defined beans, and are typically contained inside a <script> element. When this element is the root of the XML document, the BML script corresponds to a final application, that is, cannot be reused as a component (it can be reused through macro expansion as we explain later).

Recursive composition requires additional language support to define the constructor, properties, methods and events of the new bean using compositions of beans. This section describes the creation of new JavaBeans with BML. In the next section we present the related function of macro expansion in BML, which is way of reusing preconfigured BML scripts.

The example in Figure 2 shows how the Juggler application of Figure 1 can be wrapped in a bean. In this particular case, almost all the code from Figure 1 has been included as the constructor of the new bean, while two method calls have been exposed as methods of the composition.

A new component type is defined in BML using the <beanDef> element. The class for the new component is derived from the *name* attribute. The constructor, properties, methods and events of the component are defined using the <constructorDef>, <propertyDef>, <methodDef>, and <eventDef> elements respectively. These definitions can in general be provided in two ways: by *delegation* or by direct *implementation.*

When delegation is used, the composite's property, method or event is mapped to a property, method or event of a bean which is part of the composition. For example, in lines 59 to 66 of Figure 3 methods, properties and events of the composite bean are define by delegating to the *frame Juggler*, *start* and *stop* beans. In all these cases, the *name* attribute gives the name of the new method, property or event in the composite, the *sourceBean* attribute is used to identify the delegation bean, and *method*, *property* and *event* identify the method, property and event in the source bean.

When using direct implementation, the implementation is specified by a nested <script> element containing a regular BML script (which includes no bean definition elements). An example of this is provided by the constructor in Figure 3, which includes most of the code from Figure 2. Constructors are defined using a <constructorDef> element, and can only be defined by direct implementation, never by delegation.

Constructors are different from other bean elements in another important aspect. The naming scope defined by the (top level) <script> element in a constructor's definition is considered global for the complete bean definition. That is, the identifiers introduced in this script are visible everywhere in the bean. This is used, for instance, in the identification of the beans used in all definitions by delegation. In the examples from figure 2, the names specified by all the *sourceBean* attributes correspond to beans that were registered in the constructor's script. The identifiers of beans defined in the implementation of

```
0     <?xml version="1.0"?>
1
2     <beanDef name="CompositeJuggler">
3       <constructorDef>
4         <script language="bml">
5           <bean class="java.awt.Frame" id="frame">
6             <property name="title" value="IBM Juggler"/>
7             <event-binding name="window" filter="windowClosing">
8               <script>
9                 <call-method target="class:java.lang.System" name="exit">
10                  <cast class="int" value="0"/>
11                </call-method>
12              </script>
13            </event-binding>
15            <add>
16              <bean class="demos.juggler.Juggler" id="Juggler"/>
17              <string value="Center"/>
18            </add>
19            <add>
20              <bean class="java.awt.Button" id="start">
22                <property name="label" value="Start"/>
23                <event-binding name="action">
24                  <script>
25                    <call-method target="Juggler" name="start"/>
26                  </script>
27                </event-binding>
28              </bean>
29              <string value="North"/>
30            </add>
41            <add>
42              <bean class="java.awt.Button" id="stop">
43                <property name="label" value="Stop"/>
44                <event-binding name="action">
45                  <script>
46                    <call-method target="Juggler" name="stop"/>
47                  </script>
48                </event-binding>
49              </bean>
50              <string value="South"/>
51            </add>
53            <call-method name="pack"/>
54          </bean>
55          <call-method target="Juggler" name="start"/>
56        </script>
57      </constructorDef>
58
59      <methodDef name="show" sourceBean="frame" method="show"/>
60      <methodDef name="start" sourceBean="Juggler" method="start"/>
61      <methodDef name="stop" sourceBean="Juggler" method="stop"/>
62
63      <propertyDef name="startLabel" sourceBean="start" property="label"/>
64      <propertyDef name="stopLabel" sourceBean="stop" property="label"/>
65
66      <eventDef name="window" sourceBean="frame" event="window"/>
67
68    </beanDef>
```

Figure 3: The Juggler Bean

methods, properties of events are not visible outside their own script block.

Macro Expansion

BML provides a form of macro expansion that allows reusing existing BML scripts, which can then be embedded and further configured on new scripts.

To achieve this BML allows using the name of a BML file as the value of the class name attribute in the <bean> element used to instantiate the component. The nested BML file is evaluated recursively within a new scope of its own, and the resulting bean is then used as the default target bean for further composition operations.

Consider this example:

```
<bean class='redbutton.bml'>
  <property name='label'
            value='Red Button'/>
  ...
</bean>
```

where redbutton.bml is:

```
<bean class='java.awt.Button'>
  <property name='background'
            value='0xff0000'/>
</bean>
```

In this example the first BML script takes the bean produced by evaluating redbutton.bml and then sets its label property. The file redbutton.bml takes a Button component and sets its background color property to red and returns it. This simple example illustrates how a nested BML script can be used as defining a component which is then further configured and composed.

This approach amounts to macro expansion without parameterization. BML in fact allows parameterization of such scripts: the recursive invocation can be given arguments similar to how constructor arguments are given. The nested script can then retrieve the arguments and use them as it wishes. This allows the nested script to effectively be a template composition, with key parts filled in by the values of the parameters.

This type of parameterized macro expansion is not true recursive composition because we can only manipulate the features of the returned component and not of an entire composition. See the previous section for a description of how recursive composition is supported in BML.

4 Implementing BML

We have implemented two BML language processors: an interpreter and a compiler. Both implementations are designed to be embeddable and provide full access to the assembly-time environment as well as to the run-time environment. The assembly-time environment has been pre-populated with a collection of type converters and adders that provide commonly used type conversions and aggregation capabilities, respectively. These can be augmented by the host of the BML processor by accessing the environment and adding new capabilities to the registry.

The interpreter receives the BML document as an XML tree and functions based on whether the outermost element is a *beanDef* element or a *bean*/< *script* element. In the latter case, it uses Java reflection to implement the composition operations. In the case of *beanDef*, the interpreter will use the compiler to compile the bean definition upon first use and then reuse the resulting class.

The BML interpreter implements static scoping by performing name registration and resolution against the object registry in scope. Each < *script* > element introduces a new scope by creating a new registry which cascades upwards to the scope that embeds it. For event handler scripts ("glue" code), which are scripts whose execution is deferred until run-time, static scoping is achieved by storing the statically scoped registry with the script to be run at run-time. For example, the event script in line 22 of the example shown in Figure 2 binds statically to the registry in scope at assembly-time and then at run-time uses it to locate the "Juggler" component.

The compiler receives the BML script as an XML tree and uses Java reflection to generate the appropriate Java source code to implement the composition operations. If the outermost element is not *beanDef*, the compiler places the resulting code in a `main()` method. Otherwise, the bean definition

guides the target code generated.

The previously identified "assembly–time" phase for such generated composition code hence occurs at the startup of the execution of the generated code. The compiler allows one to generate code that is independent of the BML environment at assembly–time. That is, it can resolve type converters and adders as well as scoping at compile time if possible so that the generated code is straight Java code. If one wishes to have full embeddability of the generated code with the BML assembly–time environment, then it is necessary to generate code which BML dependent.

Implementing Event Bindings to Scripts

BML supports writing arbitrary scripts to be run as "glue" code. This is supported for any type of event thrown by any bean and is implemented with event adaptors, event processors and the event adaptor registry. The model consists of an event specific adaptor that receives the event from the source, delegates it to a generic event processor which then runs the script. This approach is a decomposition of the standard JavaBeans event binding model to allow dynamic look up and/or generation of event adapters.

Event adapters must implement a simple interface that is capable of receiving a handle to an event processor. An event adapter must be implemented and available from the event adapter registry for each event listener type. When the BML processor creates an event adapters and adds it as a listener to an event source, it tells the adapter what event processor to delegate the event to. Event processors are the entry point to the BML runtime and are responsible for delivering the event to the intended recipient script.

When an event adapter receives an event from an event source, it delegates the event to its event processor. The interpreter uses an event processor that actually delivers the event to a script and runs the script. The compiler can generate both customized event processors that perform this task, and customized event adapters that bypass the event processor mechanism entirely and directly deliver the event to the user's script.

The event adapter registry provides registration and lookup service for event adapters. We have also implemented the ability to on–demand generate event adapters in Java bytecode form. This eliminates the need to hand–write event adapters in many cases. (It is not possible sometimes because of security constraints of the runtime location; loading dynamically constructed classes is not always permitted.)

5 Future Work

Support Other Component Models

An important objective of the BML project is to achieve wide acceptance in the software development community by supporting other component models.

The challenge is to develop a a common component composition language supporting composition operations for the three major component models, JavaBeans, COM, and CORBA, in such a way that it can work with components from different component models in the same application. This work clearly depends on the availability of run–time bridges to go between the different models. Some of those already exist.

Concurrency Support

BML has not dealt with the issue of object concurrency. If components can be objects, and there is the possibility of concurrent execution, concurrency control can become a major issue. This is particularly important when communication channels are event streams, as in the JavaBEans model, since event handling is a common cause of race conditions and deadlocks in multithreaded environments (see [20]).

The question is whether, in these circumstances, the composition language should assure correct synchronization among components. When components from third party authors are used (maybe from several of them) and concurrent execution is necessary, it may become impossible to predict correct synchronization of the composition, and a positive answer would seem appropriate. This is the view expressed in [13], which underlies the design of the Piccola language.

The issue for BML is whether it is possible to integrate concurrency control while still keep the simplicity and transparency of the language. We have no answer to this yet.

6 Conclusions

We have presented an alternate approach to component composition in the form of a new composition language for the composition of JavaBeans components, the Bean Markup Language (BML). BML is a declarative language that uses the XML syntax to reduce the adoption barrier of both developers and machines. The language constructs are few and simple, and are designed to capture in a first–class manner the semantics of component composition. In spite its simplicity, BML provides support for most major composition operations. In particular, BML supports recursive composition, which assures scalability and enables top–down design methodologies.

BML allows composers to author event filtering scripts in arbitrary scripting languages, which opens up JavaBeans component composition to non–Java programmers. That is an important points that reinforces the notion that is not necessarily a programmer's job.

Still, BML does not address some relevant issues. One is concurrency control, which can be a critical issue in multiprocess environments. Finally, the objective of using BML as a vehicle to extend component oriented development equires that other component models be supported, if possible through a common composition language.

References

[1] F. Achermann, M. Lumpe, J.-G. Schneider, and O. Nierstrasz. Piccola - a small composition language. In H. Bowman and J. D. (Eds.), editors, *Formal Methods for Distributed Processing, an Object Oriented Approach.* Cambridge University Press, 2000, to appear.

[2] M. Aoyama. New age of software development: How component-based software engineering changes the way of software development. In *Proc. 1998 International Workshop on Component-Based Software Engineering*, 1998. http://www.sei.cmu.edu/cbs/icseworkshop.htm.

[3] U. Assmann. *Invasive Software Composition with Program Transformation.* Forthcoming habilitation, 2000.

[4] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The genvoca model of software-system generators. *IEEE Softw.*, 11(5), Sept. 1994.

[5] D. Box. *Essential COM.* Addison-Wesley, Reading, M, 1998.

[6] A. Brown. From component infrastructure to component-based development. In *Proc. 1998 International Workshop on Component-Based Software Engineering*, 1998. http://www.sei.cmu.edu/cbs/icseworkshop.htm.

[7] A. Brown and K. C. Wallnau. The current state of cbse. *IEEE Software*, September 1998.

[8] D. Flanagan. *JavaScript: The Definitive Guide.* O'Reilley, Cambridge, MA, 1998.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software.* Addison Wesley, Reading, MA, 1995.

[10] IBM Corporation. *Building Object-Oriented Frameworks.* http://www.ibm.com/java/education/oobuilding.

[11] K. L. Kroeker. Software [r]evolution: A roundtable. *IEEE Computer*, 32(5), 1999.

[12] R. Light. *Presenting XML.* Sams.Net, Indianapolis, IN, 1997.

[13] O. Nierstrasz and T. D. Meijler. Requirements for a composition language. In O. N. Paolo Ciancarini and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, Lecture Notes in Computer Science 924, Berlin, 1995. Springer.

[14] O. Nierstrasz and T. D. Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2), June 1995.

[15] O. Nierstrasz and D. Tsichritzis. *Object Oriented Software Composition.* Prentice Hall, Englewood Cliffs, NJ, 1995.

[16] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, MA, 1994.

[17] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3), 1998.

[18] J. Siegel. *CORBA: Fundamentals and Programming.* John Wiley, New York, 1996.

[19] Y. Smaragdakis and D. Batory. Scoping constructs for software generators. In *Proc. First Symposium on Generative and Component-Based Software Engineering*, September 1999.

[20] Sun Microsystems. *JavaBeans*, 1997.

[21] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, Harlow, England, 1998.

[22] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming PERL.* O'Reilley, Cambridge, MA, 1996.

[23] S. Weerawarana and M. Duftler. Bml v2.3 user's guide. *Available as a part of BML v2.3*, 1999. http://www.alphaWorks.ibm.com/formula/bml.

[24] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0 (Second Edition)*, October 2000. http://www.w3.org/TR/2000/REC-xml-20001006.

[25] World Wide Web Consortium. *XML Schema Part 1: Structures*, October 2000. http://www.w3.org/TR/xmlschema-1.

Design Patterns for Generic Programming in C++

Alexandre Duret-Lutz, Thierry Géraud, and Akim Demaille
EPITA Research and Development Laboratory
14-16 rue Voltaire, F-94276 Le Kremlin-Bicêtre cedex, France
{Alexandre.Duret-Lutz,Thierry.Geraud,Akim.Demaille}@lrde.epita.fr
http://www.lrde.epita.fr/

Abstract

Generic programming is a paradigm whose wide adoption by the C++ community is quite recent. In this scheme most classes and procedures are parameterized, leading to the construction of general and efficient software components. In this paper, we show how some design patterns from Gamma *et al.* can be adapted to this paradigm. Although these patterns rely highly on dynamic binding, we show that, by intensive use of parametric polymorphism, the method calls in these patterns can be resolved at compile-time. In intensive computations, the generic patterns bring a significant speed-up compared to their classical peers.

1 Introduction

This work has its origin in the development of Olena [11], our image processing library. When designing a library, one wants to implement algorithms that work on a wide variety of types without having to write a procedure for each concrete type. In short, one algorithm should be *generic* enough to map to a single procedure. In object-oriented programming this is achieved using *abstract types. Design Patterns*, which are design structures that have often proved to be useful in scientific computing, rely even more on abstract types and inclusion polymorphism[1].

However, when it comes to numerical computing, object-oriented designs can lead to a *huge performance loss*, especially as there may be a high number of virtual functions calls [7] required to perform operations over an abstraction. Yet, rejecting design patterns for the sake of efficiency seems radical.

In this paper, we show that some design patterns from Gamma *et al.* [10] can be adapted to generic programming. To this aim, virtual functions calls are avoided by replacing inclusion polymorphism by parametric polymorphism.

This paper presents patterns in C++, but, although they won't map directly to other languages because "genericity" differs from language to language, our work does not apply only to C++: our main focus is to devise flexible designs in contexts where efficiency is critical. In addition, C++ being a multi-paradigm programming language [28], the techniques described here can be limited to critical parts of the code dedicated to intensive computation.

In section 2 we introduce generic programming and present its advantages over classical object-oriented programming. Then, section 3 presents and discusses the design of the following patterns: GENERIC BRIDGE, GENERIC ITERATOR, GENERIC ABSTRACT FACTORY, GENERIC TEMPLATE METHOD, GENERIC DECORATOR, and GENERIC VISITOR. We conclude and consider the perspectives of our work in section 4.

2 Generic programming

By "generic programming" we refer to a use of parameterization which goes beyond simple genericity on data types. Generic programming is an abstract and efficient way of designing and assembling components [15] and interfacing them with algorithms.

[1] *Inclusion polymorphism* corresponds to virtual member functions in C++, deferred functions in Eiffel, and primitive functions in Ada.

Generic programming is an attractive paradigm for scientific numerical components [12] and numerous libraries are available on the Internet [22] for various domains: containers, graphs, linear algebra, computational geometry, differential equations, neural networks, visualization, image processing, etc.

The most famous generic library is probably the *Standard Template Library* [26]. In fact, generic programming appeared with the adoption of STL by the C++ standardization committee and was made possible with the addition of new generic capabilities to this language [27, 21].

Several generic programming idioms have already been discovered and many are listed in [30]. Most generic libraries use the GENERIC ITERATOR that we describe in 3.2. In POOMA [12] — a scientific framework for multi-dimensional arrays, fields, particles, and transforms — the GENERIC ENVELOPE-LETTER pattern appears. In the REQUESTED INTERFACE pattern [16], a GENERIC BRIDGE is introduced to handle efficiently an adaptation layer which mediates between the interfaces of the servers and of the clients.

2.1 Efficiency

The way abstractions are handled in the object-oriented programming paradigm ruins the performances, especially when the overhead implied by the abstract interface used to access the data is significant in comparison with the time needed to process the data.

For example, in an image processing library in which algorithms can work on many kinds of aggregates (two or three dimensional images, graphs, etc.), a procedure that adds a constant to an aggregate may be written using the object-oriented programming paradigm as follows.

```
template< class T >
void add (aggregate<T>& input, T value)
{
  iterator<T>& iter = input.create_iterator ();
  for (iter.first(); !iter.is_done(); iter.next())
    iter.current_item () += value;
}
```

Here, `aggregate<T>` and `iterator<T>` are abstract classes to support the numerous aggregates available: parameterization is used to achieve genericity on pixel types, and object-oriented abstractions are used to get genericity on the image structure.

	dedicated C	classical C++	generic C++
add	10.7s	37.7s	12.4s
mean	47.3s	225.8s	57.5s

Table 1: Timing of algorithms written in different paradigms. (The code was compiled with `gcc 2.95.2` and timed on an AMD K6-2 380MHz machine running GNU/Linux.)

As a consequence, for each iteration the direct call to `T::operator+=()` is drowned in the virtual calls to `current_item()`, `next()` and `is_done()`, leading to poor performances.

Table 1 compares classical object-oriented programming and generic programming and shows a speed-up factor of 3 to 4. The `add` test consists in the addition of a constant value to each element of an aggregate. The `mean` test replaces each element of an aggregate by the mean of its four neighbors. The durations correspond to 200 calls to these tests on a two dimensional image of 1024 × 1024 integers. "Dedicated C" corresponds to handwritten C specifically tuned for 2D images of integers, so the difference with classical C++ is what people call *the abstraction penalty*. While this is not a textbook case —we do have such algorithms in Olena— it is true that usually the impact of object-oriented abstraction is insignificant. High speed-ups are obtained from generic programming compared to object-oriented programming when data processing is cheap relatively to data access. For example for simple list iteration or matrix multiplication.

The generic programming writing of this algorithm, using a GENERIC ITERATOR, will be given in section 3.2.

2.2 Generic programming from the language point of view

Generic programming relies on the use of several programming language features, some of which being described below.

Genericity is the main way of generalizing object-oriented code. Not all languages support both generic classes and generic procedures (e.g., Eiffel features only generic classes).

Nested type names refers to the ability to look up a type as member of a class and allow to link related types (such as `image2d` and `iterator2d`)

together.

Constrained genericity is a way to restrict the possible values of formal parameters using signatures (e.g., when using ML functors [19]) or constraining a type to be a subclass of another (as in Eiffel or Ada 95). C++ does not provide specific language features to support constrained genericity, but subclass constraints [29, 23] or feature requirements [18, 25] can be expressed using other available language facilities [27].

Generic specialization allows the specialization of an algorithm (e.g., dedicated to a particular data type) overriding the generic implementation.

Not all languages support these features, this explains why the patterns we present in C++ won't apply directly to other languages.

2.3 Generic programming guidelines

From our experience in building Olena, which is entirely carried out by generic programming, we derived the following guidelines. These rules may seem drastic, but their appliance can be limited to critical parts of the code dedicated to intensive computation.

Guidelines for generic classes:

- Avoid inclusion polymorphism.
  In other words, the type of a variable (static type, known at compile-time) is exactly that of the instance it holds (dynamic type, known at run-time). The main requirement of generic programming is that *the concrete type of every object is known at compile-time.*
- Avoid operation polymorphism.
  Abstract methods are forbidden: dynamic binding is too expensive. Simulate operation polymorphism with either: (i) parametric classes thanks to the *Curiously Recurring Template* idiom (see section 3.4), or (ii) parametric methods, which lead to a form of *ad-hoc* polymorphism (overloading).
- Use inheritance only to factor methods and to declare attributes shared by several subclasses.

Guidelines for procedures which use generic patterns:

- Parameterize the procedures by the types of their inputs, even if the input itself is parameterized.
- Parameterize the procedures by the types of the components used (unless they can be obtained by a nested type lookup in another parameter-type).

3 Generic Design Patterns

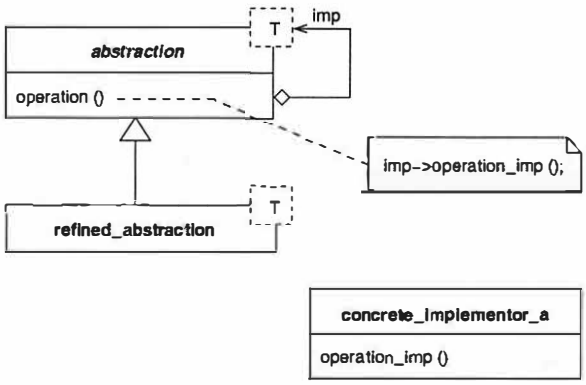
Our generic design patterns exposition is Gamma *et al.*'s description of the original, abstract version of the patterns [10]. We do not repeat the elements that can be found in this book.

3.1 Generic Bridge

Intent

Decouple an abstraction from its implementation so that the two can vary independently.

Structure



Participants

An `abstraction` class is parameterized by the `Implementation` used. Any (low-level) operation on the abstraction is delegated to the implementation instance.

Consequences

Because the implementation is statically bound to the abstraction, you can't switch implementation at runtime. This kind of restriction is common to generic programming: configurations must be known at compile-time.

Known Uses

This pattern is really straightforward and broadly used in generic libraries. For example the `allocator` parameter in STL containers is an instance of GENERIC BRIDGE.

The POOMA team [5] use the term *engine* to name implementation classes that defines the way matrices are stored in memory. This is also a GENERIC BRIDGE.

The Ada95 rational [14, section 12.6] gives an example of GENERIC BRIDGE: a generic empty package (also called signature) is used to allow multiple implementation of an abstraction (here, a *mapping*).

As in the case of the original patterns, the structure of this pattern is the same as the GENERIC STRATEGY pattern. These patterns share the same implementation.

3.2 Generic Iterator

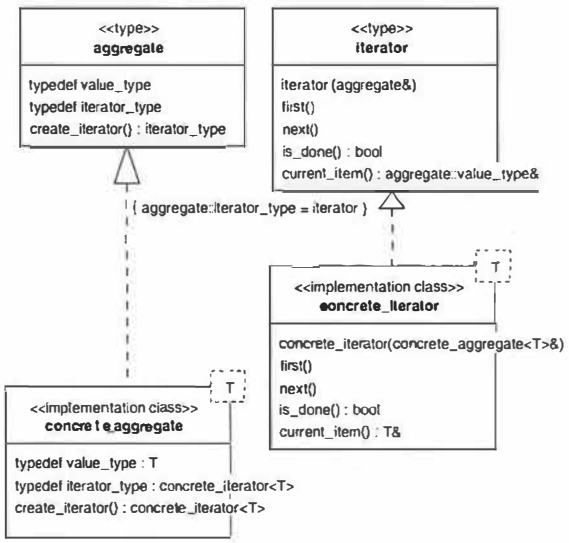
Intent

To provide an *efficient* way to access the elements of an aggregate without exposing its underlying representation.

Motivation

In numeric computing, data are often aggregates and algorithms usually need to work on several types of aggregate. Since there should be only one implementation of each algorithm, procedures must accept aggregates of various types as input and be able to browse their elements in some unified way; *iterators* are thus a very common tool. As an extra requirement compared to the original pattern, iterations must be efficient.

Structure



We use `typedef` as a non-standard extension of UML [24] to represent type aliases in classes.

Participants

The term *concept* was coined by M. H. Austern [1], to name a set of requirements on a type in STL. A type which satisfies these requirements is a *model* of this concept. The notion of concept replaces the classical object-oriented notion of abstract class.

For this pattern, two concepts are defined: *aggregate* and *iterator*, and two concrete classes model these concepts.

Consequences

Since no operation is polymorphic, iterating over an aggregate is more efficient while still being generic. Moreover, the compiler can now perform additional optimizations such as inlining, loop unrolling and instruction scheduling, that virtual function calls hindered.

Efficiency is a serious advantage. However we lose the dynamic behavior of the original pattern. For example we cannot iterate over a tree whose cells do not have the same type[2].

[2] A link between an abstract aggregate and the corresponding generic procedures can be achieved using lazy compilation and dynamic loading of generic code [8].

Implementation

Although a concept is denoted in UML by the stereotype <<type>>, in C++ it does not lead to a type: a concept only exists in the documentation. Indeed the fact that concepts have no mapping in the C++ syntax makes early detection of programming errors difficult. Several tricks have been proposed to address this issue by explicitly checking that the arguments of an algorithm are models of the expected concepts [18, 25]. In Ada 95, concept requirements (types, functions, procedures) can be captured by the formal parameters of an empty generic package (the *signature* idiom) [9].

For the user, a type-parameter (such as `Aggregate_Model` in the sample code) represents a model of *aggregate* and the corresponding model of *iterator* can then be deduced statically.

Sample Code

```
template< class T >
class buffer
{
  public:
    typedef T data_type;
    typedef buffer_iterator<T> iterator_type;
  // ...
};
```

```
template< class Aggregate_Model >
void add(Aggregate_Model& input,
         typename Aggregate_Model::data_type value)
{
 typename Aggregate_Model::iterator_type&
  iter = input.create_iterator ();

 for (iter.first(); !iter.is_done(); iter.next())
  iter.current_item () += value;
}
```

Known Uses

Most generic libraries, such as STL, use the GENERIC ITERATOR.

Variations

We translated the Gamma *et al.* version, with methods `first()`, `is_done()`, and `next()` in the iterator class. STL uses another approach where pointers should also be *models* of iterators: as a consequence, iterators cannot have methods and most of their operators will rely on methods of the container's class. This makes implementation of multiple schemes of iteration difficult: for example compare a forward and a backward iteration in STL:

```
container::iterator i;
for (i = c.begin(); i != c.end(); ++i)
     // ...

container::reverse_iterator i;
for (i = c.rbegin(); i != c.rend(); ++i)
     // ...
```

First, the syntax differs. From the STL point of view this is not a serious issue, because iterators are meant to be passed to algorithms as instances. For a wider use, however, this prevents parametric selection of the iterator (i.e., passing the iterator as a type). Second, you have to implement as many `xbegin()` and `xend()` methods as there are schemes of iteration, leading to a higher coupling [17] between iterators and containers.

Another idea consists in the removal of all the iterator related definitions, such as `create_iterator()` or `iterator_type`, from `concrete_aggregate<T>` in order to allow the addition of new iterators without modifying the existing aggregate classes [32]. This can be achieved using *traits classes* [20] to associate iteration schemes with aggregates: the iterated aggregate instance is given as an argument to the iterator constructor. For example we would rewrite the `add()` function as follows.

```
template< class Aggregate_Model >
void add(Aggregate_Model& input,
         typename Aggregate_Model::data_type value)
{
 typename forward_iterator< Aggregate_Model >::type
  iter (input);

 for (iter.first(); !iter.is_done(); iter.next())
  iter.current_item () += value;
}
```

This eliminates the need to declare iterators into the aggregate class, and allows further additions of iteration schemes by the simple means of creating a new traits class (for example `backward_iterator<T>`).

3.3 Generic Abstract Factory

Intent

To create families of related or dependent objects.

Motivation

Let us go back over the different iteration schemes problem discussed previously. We want to define several kind of iterators for an aggregate, and as so we are candidates for the ABSTRACT FACTORY pattern. The STL example can be rewritten as follows to make this pattern explicit: iterators are *products*, built by an aggregate which can be seen as a *factory*.

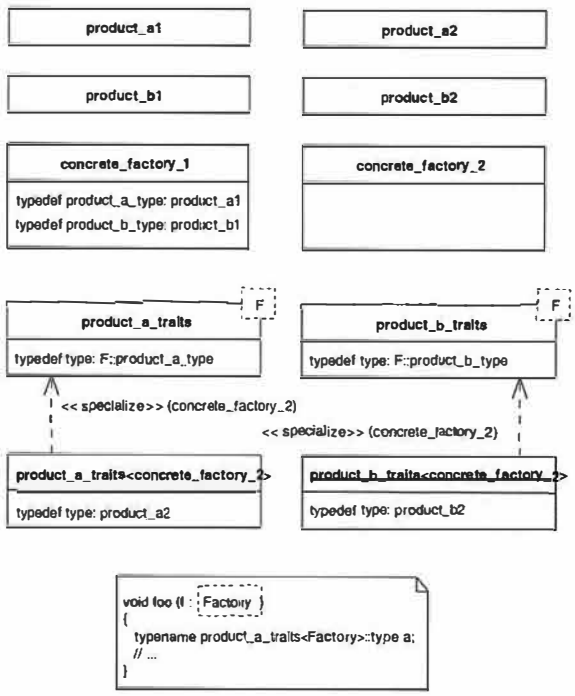
```
factory_a::product_1 i;
for (i = c.begin(); i != c.end(); ++i)
     // ...

factory_a::product_2 i;
for (i = c.rbegin(); i != c.rend(); ++i)
     // ...
```

Implementing a GENERIC ABSTRACT FACTORY is therefore just a matter of defining the product types in the classes that should be used as a factory. This is really simpler than the original pattern. Yet there is one significant difference in usage: an ABSTRACT FACTORY returns an *object* whereas a GENERIC ABSTRACT FACTORY returns a *type*, giving more flexibility (e.g. constructors can be overloaded).

We have shown that if we want to implement multiple iteration schemes, it is better to use traits classes, to define the schemes out of the container. A trait class if a GENERIC ABSTRACT FACTORY too (think of `trait::type` as `factory::product`). But one issue is that these two techniques are not homogeneous. Say we want to add a new iterator to the STL containers: we cannot change the container classes, therefore we define our new iterator in a traits, but now we must use a different syntax whether we use one iterator or the other.

The structure we present here takes care of this: both internal and external definitions of products can be made, but the user will always use the same syntax.

Structure



Here, we represent a parametric method by boxing its parameter. For instance, Factory is a type-parameter of the method Accept. This does cannot conform to UML since UML lacks support for parametric methods.

Participants

We have two factories, named `concrete_factory_1` and `concrete_factory_2` which each defines two products: `product_a_type` and `product_b_type`. The first factory define the products intrusively (in it's own class), while the second do it externally (in the product's traits).

To unify the utilization, the traits default is to use the type that might be defined in the "factory" class. For example the type a defined in `foo<Factory>`, defined as `product_a_trait<Factory>::type` will equal to `concrete_factory_1::product_a_type` in the case `Factory` is `concrete_factory_1`.

Consequences

Contrary to the pattern of Gamma, inheritance is no longer needed, neither for factories, nor for products. Introducing a new product merely requires adding a new

parametrized structure to handle the types aliases (e.g., `product_c_traits`), and to specialize this structure when the alias `product_c_type` is not provided by the factory.

Known Uses

Many uses of this pattern can be found in STL. For example all the containers whose contents can be browsed forwards or backwards[3] define two products: forward and backward iterators.

The actual type of a list iterator never explicitly appears in client code, as for any class name of concrete products. Rather, the user refers to `A::iterator`, and `A` is an STL container used as a concrete factory.

3.4 Generic Template Method

Intent

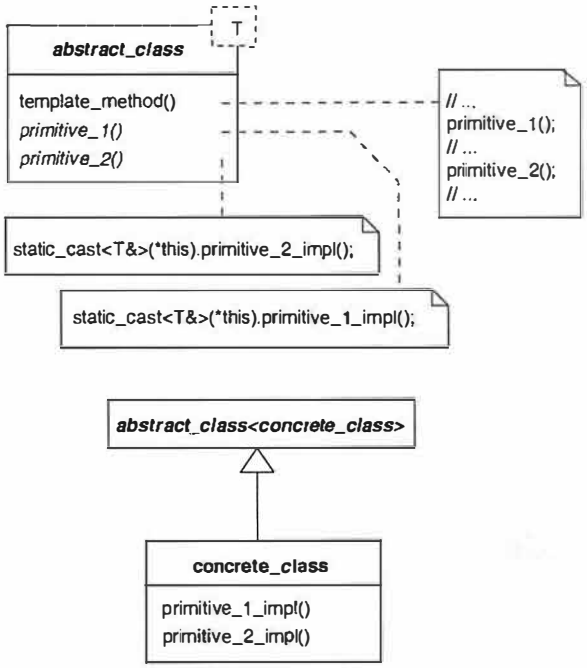
To define the canvas of an *efficient* algorithm in a superior class, deferring some steps to subclasses.

Motivation

In generic programming, we limit inheritance to factor methods [section 2.3]; here, we want a superior class to define an operation some parts of which (primitive operations) are defined only in inferior classes. As usual we want calls to the primitive operations, as well as calls to the template method, to be resolved at compile-time.

[3] vectors, doubly linked lists and dequeues are models of this concept, named *reversible containers*

Structure



Participants

In the object-oriented paradigm, the selection of the target function in a polymorphic operation can be seen as a search for the function, browsing the inheritance tree upwards from the dynamic type of the object. In practice, this is done at run-time by looking up the target in a table of function pointers.

In generic programming, we want that selection to be solved at compile-time. In other words, each caller should statically know the dynamic type of the object from which it calls methods. In the case of a superior class calling a method defined in a child class, the knowledge of the dynamic type can be given as a template parameter to the superior class. Therefore, any class needing to know its dynamic type will be parameterized by its leaf type.

The parametric class `abstract_class` defines two operations: `primitive_1()` and `primitive_2()`. Calling one of these operations leads to casting the target object into its dynamic type. The methods executed are the implementations of these operations, `primitive_1_impl()` and `primitive_2_impl()`. Because the object was cast into its leaf type, these functions are searched for in the object hierarchy from the leaf type up as desired.

When the programmer later defines the class `concrete_class` with the primitive operation implementations, the method `template_method()` is inherited and a call to this method leads to the execution of the proper implementations.

Consequences

In generic programming, operation polymorphism can be simulated by "parametric polymorphism through inheritance" and then be solved statically. The cost of dynamic binding is avoided; moreover, the compiler is able to inline all the code, including the template method itself. Hence, this design is more efficient.

Implementation

The methods `primitive_1()` and `primitive_2()` do not contain their implementation but a call to an implementation; they can be considered as *abstract methods*. Please note that they can also be called by the client without knowing that some dispatch is performed.

This design is made possible by the typing model used for C++ template parameters. A C++ compiler has to delay its semantic analysis of a template function until the function is instantiated. The compiler will therefore accept the call to `T::primitive_1_impl()` without knowing anything about `T` and will check the presence of this method later when the call to the `A<T>::primitive_1()` is actually performed, if it ever is. In Ada [13], on the contrary, such postponed type checking does not exist, for a function shall type check even if it is not instantiated. This pattern is therefore not applicable *as is* in this language.

One disadvantage of this pattern over Gamma's implementation is directly related to this: the compiler won't check the actual presence of the implementations in the subclasses. While a C++ compiler will warn you if you do not supply an implementation for an abstract function, even if it is not used, that same compiler will be quiet if pseudo-virtual operations like `primitive_1_impl()` are not defined and not used. Special care must thus be taken when building libraries not to forget such functions since the error won't come to light until the function is actually used.

We purposely added the suffixes `_impl` to the name of primitives to distinguish the implementation functions. One could image that the implementation would use the same name as the primitive, but this require some additional care as the abstract primitive can call itself recursively when the implementation is absent.[4]

Sample Code

The following code shows how to define a `get_next()` operation in each iterator of a library of containers. Obviously, `get_next()` is a template method made by issuing successive calls to the `current_item()` and `next()` methods of the actual iterator.

We define this method in a superclass `iterator_common` parametrized by its subtype, and have all iterators derive from this class.

```
template< class Child, class Value_Type >
class iterator_common
{
 public:
  Value_Type& get_next () {
   // template method
   Value_Type& v = current_item ();
   next ();
   return v;
  }
  Value_Type& current_item () {
   // call the actual implementation
   static_cast<Child&>(*this).current_item_impl();
  }
  void next () {
   // call the actual implementation
   static_cast<Child&>(*this).next_impl();
  }
};


// sample iterator definition
template< class Value_Type >
class buffer_iterator: public
  iterator_common< buffer_iterator< Value_Type >,
                   Value_Type >
{
  public:
    Value_Type current_item_impl () { ... };
    void next_impl () { ... };
    void first () { ... };
    void is_done () { ... };
    // ...
};
```

Known Uses

This pattern relies on an idiom called *Curiously Recurring Template* [4] derived from the *Barton and Nackman*

[4] You can ensure at compile-time that two functions (the primitive and its implementation) are different by passing their addresses to a helper template specialized in the case its two arguments are equal.

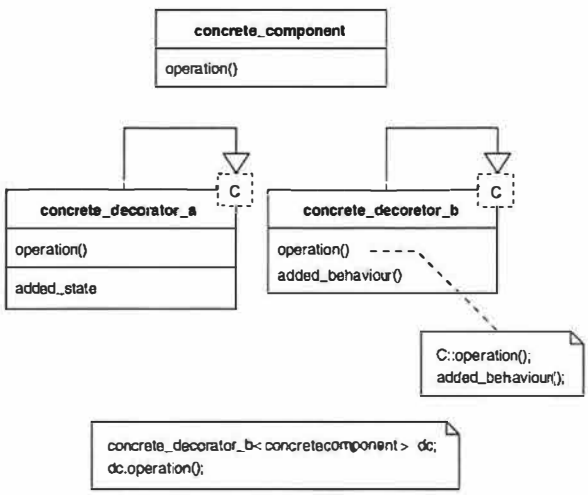
Trick [2]. In [2] this idiom is used to define a binary operator (for instance +) in a superior class from the corresponding unary operator (here +=) defined in an inferior class. Further examples are given in [30].

3.5 Generic Decorator

Intent

To *efficiently* define additional responsibilities to a set of objects or to replace functionalities of a set of objects, by means of subclassing.

Structure



We use a special idiom: having a parametric class that derives from one of its parameters. This is also known as *mixin inheritance*[5] [3].

Participants

A class `concrete_component` which can be decorated, offers an operation `operation()`. Two parametric decorators, `concrete_decorator_a` and `concrete_decorator_b`, whose parameter is the decorated type, override this operation.

[5] Mixins are often used in Ada to simulate multiple inheritance [14].

Consequences

This pattern has two advantages over Gamma's. First, any method that is not modified by the decorator is automatically inherited. While Gamma's version uses composition and must therefore delegate each unmodified operation. Second, decoration can be applied to a set of classes that are not related via inheritance. Therefore, a decorator becomes truly generic.

On the other hand we lose the capability of dynamically adding a decoration to an object.

Sample Code

Decorating an iterator of STL is useful when a container holds structured data, and one wants to perform operations only on a field of these data. In order to access this field, the decorator redefines the data access operator `operator*()` of the iterator.

```
// A basic red-green-blue struct
template< class T >
struct rgb
{
  typedef T red_type;
  red_type red;

  typedef T green_type;
  green_type green;

  typedef T blue_type;
  blue_type blue;
};


// An accessor class for the red field.
template< class T >
class get_red
{
  public:
    typedef T input_type;
    typedef typename T::red_type output_type;

    static output_type&
    get (input_type& v) {
      return v.red;
    }

    static const output_type&
    get (const input_type& v) {
      return v.red;
    }
};
```

Note how the `rgb<T>` structure exposes the type of each attribute. This makes cooperation between objects easier: here the `get_red` accessor will look up the `red_type` type member and doesn't have to know that fields of `rgb<T>` are of type `T`. `get_red` can therefore

apply to any type that features `red` and `red_type`, it is not limited to `rgb<T>`.

```
// A decorator for any iterator
template< class Decorated,
          template< class > class Access >
class field_access: public Decorated
{
 public:
  typedef typename Decorated::value_type value_type;
  typedef Access< value_type >           accessor;
  typedef typename accessor::output_type output_type;

  field_access () : Decorated () {}
  field_access (const Decorated& d) : Decorated (d) {}

  // Overload operator*, use the given accessor
  // to get the proper field.
  output_type& operator* () {
    return accessor::get (Decorated::operator* ());
  }

  const output_type& operator* () const {
    return accessor::get (Decorated::operator* ());
  }
};
```

`field_access` is a decorator whose parameters are the types of the decorated iterator, and of a helper class which specifies the field to be accessed. Actually, this second parameter is an example of the GENERIC STRATEGY pattern [6, 30].

```
int main ()
{
  typedef std::list< rgb< int > > A;
  A input;
  // ... initialize the input list ...

  // Build decorated iterators.
  field_access< A::iterator, get_red >
    begin = input.begin (),
    end   = input.end ();
  // Assign 10 to each red field.
  std::fill (begin, end, 10);
}
```

The `std::fill()` procedure is a standard STL algorithm which assigns a value to each element of a range (specified by two iterators). Since `std::fill()` is here given decorated iterators it will only assign red fields to 10.

Note that the decorator is independent of the decorated iterator: it can apply to any STL iterator, not only `list<T>::iterator`. The `std::fill()` algorithm will use methods of `field_access` inherited from the decorated iterator, such as the assignment, comparison, and pre-increment operators.

Known Uses

Parameterized inheritance is also called *mixin inheritance* and is one way to simulate multiple inheritance in Ada 95 [14]. This can also be used as an alternate way for providing *template methods* [6].

3.6 Generic Visitor

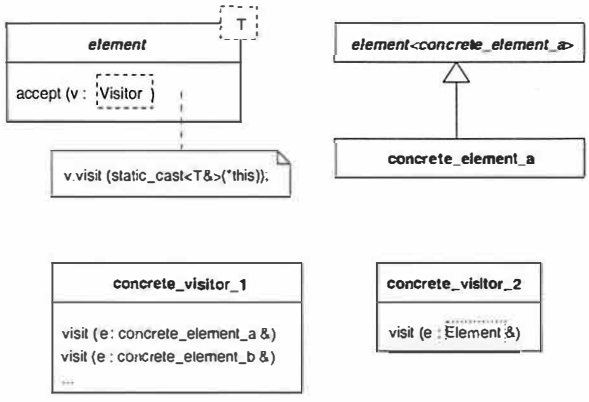
Intent

To define a new operation for the concrete classes of a hierarchy without modifying the hierarchy.

Motivation

In the case of the VISITOR pattern, the operation varies with the type of the operation target. Since we assume to know the exact type as compile-time, a trivial design is thus to define this operation as a procedure overloaded for each target. Such a design, however, does not have the advantages of the translation of the VISITOR pattern proposed in the next section.

Structure



Participants

In the original Gamma's pattern the method `accept` has to be defined in each `element`. The code of each of these `accept` method can be the same[6], only type of the

[6]This is not actually the case in Gamma's book, because the name of the visiting method to call is dependent on the element type; how-

`this` pointer changes. Here we use the same trick as the GENERIC TEMPLATE METHOD to factor `accept` in the superclass.

Each visitor defines a method `visit`, for each element type that it must handle. `visit` can be either an overloaded function (as in `concrete_visitor_1`) or a function template (as in `concrete_visitor_2`). In both case, the overload resolution or function instantiation is made possible by the exact knowledge of the element type.

One advantage of using a member template (as in `concrete_visitor_2`), over an overloaded function (as in `concrete_visitor_1`) is that the `concrete_visitor_2` class does need to be changed when new type are added: the visitor can be specialized externally should the default be inadequate.

Consequences

The code is much closer to the one of Gamma than the trivial design presented before, because the visitor is here an object with all its advantages (state, life duration).

While `accept` and `visit` does not return anything in the original pattern, they can be taught to. It the GENERIC ITERATOR they can even return a type dependent on the visitor's type. As the following example shows.

Sample Code

Let's consider an image2d class the pixels of which should be addressable using different kind of positions (Cartesian or polar coordinates, etc.). For better modularity, we don't want the image2d to known all position types. Therefore we see positions as visitors, which the image accepts. `accept` returns the pixel value corresponding to the supplied position. The image will provide only one access method, and it is up to the visitor to perform necessary conversion (e.g. polar to Cartesian) to use this interface.

A position may also refer to a particular channel in a color image. The `accept` return type is thus dependent on the visitor. We will use a traits to handle this.

ever, using the same name (`visit`) for all these methods make no problem in any language as C++ which support function overloading.

```
template< class Visitor, class Visited >
struct visit_return_trait;
```

For each pair (`Visitor`, `Visited`) `visit_return_trait<Visitor,Visited>::type` is the return type of access and visit.

```
// factor the definition of accept for all images
template < class Child >
class image {
public:
  template < typename Visitor >
  typename visit_return_trait< Visitor, Child >::
  type accept (Visitor& v) {
    return v.visit (static_cast< Child& > (*this));
  }
  // ... likewise for const accept
};
```

```
template< typename T >
class image_2d : public image< image_2d< T > > {
public:
  typedef T pixel_type;
  // ...
  T& get_value (int row, int col){...}
  const T& get_value const (int row, int col){...}
};
```

Here is one possible visitor, with it's corresponding `visit_return_trait` specialization.

```
class point_2d {
public:
  point_2d (int row, int col) { ... }

  template < typename Visited >
  typename Visited::pixel_type&
  visit (Visited& v) {
    return v.get_value (row, col);
  }
  // ...
  int row, col;
};

template< class Visited >
struct visit_return_trait< point_2d, Visited > {
  typedef typename Visited::pixel_type type;
};
```

`channel_point_2d` is another visitor, which must be parametered to access a particular layer (as in the decorator example).

```
template< template< class > class Access >
class channel_point_2d {
public:
  channel_point_2d (int row, int col) { ... }

  template < typename Visited >
  typename Access< typename Visited::pixel_type >::
  output_type& visit (Visited& v) {
    return Access< typename Visited::pixel_type >::
      get (v.get_value (row, col));
  }
  // ...
};

template< template< class > class Access,
          class Visited >
struct visit_return_trait
  < channel_point_2d< Access >, Visited > {
  typedef typename
    Access< typename Visited::pixel_type >::
      output_type type;
};
```

Finally, the following hypothetical main shows how the return value of `accept` differ according to the visitor used.

```
int main () {
  image_2d< rgb< int > > img;
  point_2d p(1, 2);
  channel_point_2d<get_red> q(3, 4);

  int v       = img.accept (p);
  rgb<int> w = img.accept (q);
}
```

In our library, `accept` and `visit` are both named `operator[]` so we can write `img[p]` or `p[img]` at will.

4 Conclusion and Perspectives

Based on object programming, generic programming allows to build and assemble reusable components [15] and proved to be useful where efficiency is required.

Since generic programming (or more generally *Generative programming* [31, 6]) is becoming more popular and because much experience and knowledge have been accumulated and assimilated in structuring the object-oriented programming, we believe that it is time to explore the benefits that the former can derive from well-proven designs in the latter.

We showed how design patterns can be adapted to the generic programming context by presenting the generic versions of three fundamental patterns from Gamma *et al.* [10]: the GENERIC BRIDGE, GENERIC ITERATOR, the GENERIC ABSTRACT FACTORY, the GENERIC TEMPLATE METHOD, the GENERIC DECORATOR, and the GENERIC VISITOR. We hope that such work can provide some valuable insight, and aid design larger systems using generic programming.

Acknowledgments

The authors are grateful to Philippe Laroque for his fruitful remarks about the erstwhile version of this work, to Colin Shaw for his corrections on an early version of this paper, and to Bjarne Stroustrup for his valuable comments on the final version.

Availability

The source of the patterns presented in this paper, as well as other generic patterns, can be downloaded from `http://www.lrde.epita.fr/download/`.

References

[1] Matthew H. Austern. *Generic programming and the STL – Using and extending the C++ Standard Template Library*. Professional Computing Series. Addison-Wesley, 1999.

[2] John Barton and Lee Nackman. *Scientific and engineering C++*. Addison-Wesley, 1994.

[3] Gilad Bracha and William Cook. Mixin-based inheritance. In *procdings of ACM Conference on Object-Oriented Programming: System, Languages, and APPLICATION (OOPSLA) 1990*. ACM, 1989. URL `http://java.sun.com/people/gbracha/oopsla90.ps`.

[4] James Coplien. Curiously recurring template pattern. In Stanly B. Lippman, editor, *C++ Gems*. Cambridge University Press & Sigs Books, 1996. URL `http://people.we.mediaone.net/stanlipp/gems.html`.

[5] James A. Crotinger, Julian Cummings, Scott Haney, William Humprey, Steve Karmesin, John Reynders, Stephen Smith, and Timothy J. Williams. Generic programming in POOMA and PETE. In *Dagstuhl seminar on Generic Programming*, April 1998. URL

http://www.acl.lanl.gov/pooma/papers/GenericProgrammingPaper/dagstuhl.pdf.

[6] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative programming. Methods, Tools, and Applications*. Addison Wesley, 2000. URL http://www.generative-programming.org/.

[7] Karel Driesen and Urs Hölzle. The Direct Cost of Virtual Function Calls in C++. In *11th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'96)*, 1996. URL http://www.cs.McGill.CA/ACL/papers/oopsla96.html.

[8] Alexandre Duret-Lutz. Olena: a component-based platform for image processing, mixing generic, generative and oo programming. In *symposium on Generative and Component-Based Software Engineering, Young Researchers Workshop*, 10 2000. URL http://www.lrde.epita.fr/publications/.

[9] Ulfar Erlingsson and Alexander V. Konstantinou. Implementing the C++ Standard Template Library in Ada 95. Technical Report TR96-3, CS Dept., Rensselaer Polytechnic Institute, Troy, NY, January 1996. URL http://www.adahome.com/Resources/Papers/General/stl2ada.ps.Z.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns – Elements of reusable object-oriented software*. Professional Computing Series. Addison Wesley, 1995.

[11] Thierry Géraud, Yoann Fabre, Alexandre Duret-Lutz, Dimitri Papadopoulos-Orfanos, and Jean-François Mangin. Obtaining genericity for image processing and pattern recognition algorithms. In *15th International Conference on Pattern Recognition (ICPR'2000)*, September 2000. URL http://www.lrde.epita.fr/publications/.

[12] Scott Haney and James Crotinger. How templates enable high-performance scientific computing in C++. *IEEE Computing in Science and Engineering*, 1(4), 1999. URL http://www.acl.lanl.gov/pooma/papers.html.

[13] *Ada95 Reference Manual*. Intermetrics, Inc., December 1994. URL http://adaic.org/standards/95lrm/. Version 6.00 (last draft of ISO/IEC 8652:1995).

[14] *Ada 95 Rationale*. Intermetrics, Inc., Cambridge, Massachusetts, January 1995. URL ftp://ftp.lip6.fr/pub/gnat/rationale-ada95/.

[15] Mehdi Jazayeri. Component programming – a fresh look at software components. In *Procedings of the 5th European Software Engineering Conference (ESEC'95)*, pages 457-478, September 1995.

[16] Ullrich Köthe. Requested interface. In *In Proceedings of the 2nd European Conference on Pattern Languages of Programming (EuroPLoP '97)*, Munich, Germany, 1997. URL http://www.riehle.org/events/europlop-1997/p16final.pdf.

[17] John Lakos. *Large-scale C++ software design*. Addison-Wesley, 1996.

[18] Brian McNamara and Yannis Smaragdakis. Static interfaces in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000. URL http://oonumerics.org/tmpw00/.

[19] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. MIT Press, 1997.

[20] Nathan C. Myers. Traits: a new and useful template technique. *C++ Report*, 7(5):32–35, June 1995. URL http://www.cantrip.org/traits.html.

[21] Nathan C. Myers. Gnarly new C++ language features, 1997. URL http://www.cantrip.org/gnarly.html.

[22] OON. The object-oriented numerics page. URL http://oonumerics.org/oon.

[23] Esa Pulkkinen. Compile-time determination of base-class relationship in C++, June 1999. URL http://lazy.ton.tut.fi/~esap/instructive/base-class-determination.html.

[24] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language – Reference manual*. Object Technology Series. Addison-Wesley, 1999.

[25] Jeremy Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000. URL http://oonumerics.org/tmpw00/.

[26] Alex Stepanov and Meng Lee. *The Standard Template Library*. Hewlett Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304, October 1995. URL `http://www.cs.rpi.edu/~musser/doc.ps`.

[27] Bjarne Stroustrup. *The Design and Evolution of C++*. Addision Wesley, June 1994. URL `http://www.research.att.com/~bs/dne.html`.

[28] Bjarne Stroustrup. Why C++ isn't just an Object-Oriented Programming Language. In *OOPSLA'95*, October 1995. URL `http://www.research.att.com/~bs/papers.html`.

[29] Petter Urkedal. Tools for template metaprogramming. web page, March 1999. URL `http://matfys.lth.se/~petter/src/more/metad/index.html`.

[30] Todd L. Veldhuizen. Techniques for scientific C++, August 1999. URL `http://extreme.indiana.edu/~tveldhui/papers/techniques/`.

[31] Todd L. Veldhuizen and Dennis Gannon. Active libraries – Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM Press, October 1998. URL `http://extreme.indiana.edu/~tveldhui/papers/oo98.html`.

[32] Olivier Zendra and Dominique Colnet. Adding external iterators to an existing Eiffel class library. In *32nd conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific'99)*, Melbourne, Australia, November 1999. IEEE Computer Society. URL `http://SmallEiffel.loria.fr/papers/tools-pacific-1999.pdf.gz`.